第 3 章

Python 基础

本章默认读者没有 Python 基础，所以讲述的 Python 知识比较详细，目的在于满足后期 UI 自动化测试框架开发的需要，如果读者已经有一定的 Python 基础，则可以自行跳过本章。如果想更详细地了解 Python 的基础知识，则建议读者购买一本 Python 基础教程进行系统学习。

3.1 Python 命名规则

开发语言中变量、方法、类都涉及命名问题，在 Python 语言中，命名会遵循一定的规则，具体如下：

(1) 命名时只能使用英文单词、下画线、数字。

(2) 命名时可以使用英文单词或下画线开头，但下画线开头有特殊的含义，不能乱用。

(3) 命名时不能与系统或第三方模块重名。

(4) 命名时类名称的首字母需要大写。

(5) 命名时需要见名知意。

3.2 Python 注释

注释用来解释变量、方法、类的含义，目的是让不熟悉代码的人可以通过注释信息快速地理解代码。Python 语言的注释分为两种，一种为单行注释，另一种为多行注释，具体如下。

(1) 单行注释：使用 # 表示。单行注释可以作为单独的一行放在被注释代码行之上，也可以放在语句或表达式之后，具体如下：

```
#这是单行注释
```

(2) 多行注释：使用 3 个单引号或 3 个双引号表示。当注释内容过多时会导致一行无法显示全部内容，此时就可以使用多行注释，具体如下：

```
"""
这是多行注释 1
"""
'''
这是多行注释 2
'''
```

注释不是为了描述代码,所以注释不是越多越好。因为一般阅读代码的人了解 Python 语法,只不过不知道代码具体要做什么事情,所以在实际开发过程中读者只需注释较为复杂或难以理解的代码。


13min

3.3 Python 变量和数据类型

变量指的是在程序运行过程中可以改变的量,也是程序在运行时临时存储数据的地方。Python 变量在定义时不需要指明数据类型,只需写明变量,然后用等号赋值。如果想知道变量的数据类型,则可以使用 type()方法对变量数据类型进行查看。


30min

3.3.1 常用变量定义

在本节中,笔者仅介绍变量的定义和简单的用法,目的是让读者对变量有个初步的认识,后面章节中笔者会针对不同变量进行详细讲解。

1. 字符串

字符串变量是应用最为广泛的变量,例如读者可以将名字赋值给字符串变量,代码如下:

```
//第 3 章/new_python/my_python_1.py
name = "栗子软件测试"
print("变量 name 类型为{}".format(type(name)))
print("变量 name 的值为{}".format(name))

#执行结果
变量 name 类型为<class 'str'>
变量 name 的值为栗子软件测试
```

示例中,笔者将名字赋值给了 name 变量,并且名字加了双引号,表示字符串。为了查看变量具体是什么类型,笔者使用 type()方法打印变量 name 的数据类型,结果为<class 'str'>,表示 name 变量是字符串类型。

另外,笔者在打印时还使用了 format()方法,该方法是字符串格式化方法。使用 format()方法时,方法中的参数会替换字符串中的{},format()方法中如果有多个参数,则需要使用逗号分隔。

2. 数字

数字变量分为整型(int)、浮点型(float)、布尔型(bool)、复数(complex),读者可以主要

关注常用的整型、浮点型、布尔型变量，代码如下：

```
//第 3 章/new_python/my_python_1.py
age = 18
score = 99.5
pass_or_not = True
print("变量 age 的类型为{}".format(type(age)))
print("变量 age 的值为{}".format(age))
print("变量 score 的类型为{}".format(type(score)))
print("变量 score 的值为{}".format(score))
print("变量 pass_or_not 的类型为{}".format(type(pass_or_not)))
print("变量 pass_or_not 的值为{}".format(pass_or_not))

#执行结果
变量 age 的类型为<class 'int'>
变量 age 的值为 18
变量 score 的类型为<class 'float'>
变量 score 的值为 99.5
变量 pass_or_not 的类型为<class 'bool'>
变量 pass_or_not 的值为 True
```

示例中，笔者定义了 age 变量，表示年龄，赋值为 18；定义了 score 变量，表示分数，赋值为 99.5；定义了 pass_or_not 变量，表示是否及格，赋值为 True。从执行结果可以看出，age 变量是整型 int；score 变量是浮点型 float；pass_or_not 变量是布尔类型 bool。

其中整型表示整数，包含 0 和正负整数；浮点型由整数部分和小数部分组成；布尔类型的值为 True 或 False，可以用来控制程序的流程，例如，如果判断条件成立(True)，则执行方法 1，如果条件不成立(False)，则执行方法 2。

3. 元组

元组元素写在圆括号中，元组可以包含一个或多个元素。如果元组有多个元素，则元素之间用逗号分隔；如果元组只有一个元素，则元素后边必须加上逗号。一般不需要改动的数据会定义在元组中，例如在配置文件中定义数据库的信息，代码如下：

```
//第 3 章/new_python/my_python_1.py
db_tuple = ('127.0.0.1', 3306, 'admin', 123456)
print("变量 db_tuple 的类型为{}".format(type(db_tuple)))
print("变量 db_tuple 中的第 1 个值为{}".format(db_tuple[0]))
print("变量 db_tuple 中的第 1 个值的类型为{}".format(type(db_tuple[0])))

#执行结果
变量 db_tuple 的类型为<class 'tuple'>
变量 db_tuple 中的第 1 个值为 127.0.0.1
变量 db_tuple 中的第 1 个值的类型为<class 'str'>
```

示例中，笔者将数据库的 IP 地址、端口号、用户名、密码放在一个元组中，其中 IP 地址和用户名是字符串类型，端口号和密码是整型，说明元组中可以包含不同的数据类型。

从执行结果可以看出，变量 db_tuple 的类型为<class 'tuple'>，表示元组。笔者还打印

了元组中的第 1 个元素的值 db_tuple[0],结果是 127.0.0.1。说明元组中元素是有序的,并且第 1 个元素的下标是 0。

那么仅包含一个元素的元组为什么需要在元素后面加逗号呢?笔者接下来简单地进行演示说明,代码如下:

```
//第 3 章/new_python/my_python_1.py
tuple_1 = ('127.0.0.1')
tuple_2 = ('127.0.0.1',)
print("变量 tuple_1 的类型为{}".format(type(tuple_1)))
print("变量 tuple_2 的类型为{}".format(type(tuple_2)))

#执行结果
变量 tuple_1 的类型为<class 'str'>
变量 tuple_2 的类型为<class 'tuple'>
```

示例中,笔者定义了两个变量 tuple_1 和 tuple_2,两个变量都写在圆括号中,并且都包含一个元素,其中 tuple_1 元素后边没有加逗号,tuple_2 元素后边加了一个逗号,笔者打印了两个变量的数据类型。从执行结果可以看出,Python 认为 tuple_1 变量是字符串,tuple_2 变量才是包含一个元素的元组。

4. 列表

列表元素写在方括号中,可以包含一个或多个元素,多个元素之间也需要用逗号分隔。跟元组一样,列表也可以包含不同类型的元素,也可以使用下标获取对应位置的元素,下标还是从 0 开始。不同的是当列表中只有一个元素时,该元素后边不需要添加逗号。

如果需要将学生信息保存到列表中,则该如何实现呢?代码如下:

```
//第 3 章/new_python/my_python_1.py
stu_list = ["栗子软件测试", 18, 99.5, True]
print("变量 stu_list 的类型为{}".format(type(stu_list)))
print("变量 stu_list 中的分数值为{}".format(stu_list[2]))
print("变量 stu_list 中的分数类型为{}".format(type(stu_list[2])))

#执行结果
变量 stu_list 的类型为<class 'list'>
变量 stu_list 中的分数值为 99.5
变量 stu_list 中的分数类型为<class 'float'>
```

示例中,笔者将前面例子中的字符串和数字类型的变量都写在方括号中,从而组成了列表,说明列表可以包含不同类型的变量。从执行结果可以看出,变量 stu_list 的类型是<class 'list'>,表示列表。笔者还打印了列表中下标为 2 的元素,打印结果是学生的分数 99.5,说明列表的下标也是从 0 开始的。

5. 集合

集合元素写在花括号中,也可以包含一个或多个元素,多个元素之间使用逗号隔开。集合中的元素是无序的,所以不能使用下标来找到集合中的元素。笔者将学生信息列表改用

集合实现，看一下 Python 会报什么错误，代码如下：

```
//第 3 章/new_python/my_python_1.py
stu_set = {"栗子软件测试", 18, 99.5, True}
print("变量 stu_set 的类型为{}".format(type(stu_set)))
print("变量 stu_set 中的分数值为{}".format(stu_set[2]))
print("变量 stu_set 中的分数类型为{}".format(type(stu_set[2])))

#执行结果
变量 stu_set 的类型为<class 'set'>
TypeError: 'set' object does not support indexing
```

示例中，笔者将学生信息写在花括号中，赋值给了 stu_set 变量。从执行结果可以看出，变量类型为<class 'set'>，表示集合。当使用下标打印集合的第 3 个元素时，系统报错'set' object does not support indexing，意思是集合不支持下标操作，所以如果读者想要通过下标获取某个元素就不能使用集合，但可以使用元组或列表。

6. 字典

字典元素也写在花括号中，但元素为 key:value 格式，读者可以根据字典的 key 获得对应的 value 值。当学生信息使用字典来保存时，读者就可以根据字典的 key 更明确地获得相应的 value 值，代码如下：

```
//第 3 章/new_python/my_python_1.py
stu_dict = {'name':'栗子软件测试', 'age':18, 'score':99.5, 'pass_or_not':True}
print("变量 stu_dict 的类型为{}".format(type(stu_dict)))
print("变量 stu_dict 中的分数值为{}".format(stu_dict['score']))

#执行结果
变量 stu_dict 的类型为<class 'dict'>
变量 stu_dict 中的分数值为 99.5
```

示例中，笔者将学生信息使用 key:value 的格式写在花括号中，赋值给了 stu_dict 变量。从执行结果可以看出，变量的类型为<class 'dict'>，表示字典。当笔者需要获取学生的分数时，只需使用 stu_dict['score']格式便可以获取分数值。

7. 总结

从以上的示例中读者会发现变量赋值非常简单，只需使用变量名＝变量值的格式，但不同变量的定义还是有一些需要注意的地方，总结如下：

(1) 布尔类型变量的值为 True 或 False。

(2) 在元组、列表、字典、集合中可以存放不同类型的值，如数字、字符串等。

(3) 字典和集合都使用花括号{}表示，但字典中存放的是键-值对形式 key:value。

除了可以正常地定义变量外，读者还需要注意如何定义空变量。例如字典和集合都用花括号表示，那么定义空字典和空集合时应该如何进行区别呢？笔者对空变量的定义简单地进行了总结，见表 3-1。

表 3-1 Python 常用空变量定义

变　量	空变量定义	备　注
字符串(sting)	str =''或 str = ""	单引号或双引号均可
元组(tuple)	my_tuple = ()	空元组用圆括号定义
列表(list)	my_list = []	空列表用方括号定义
集合(set)	my_set = set()	空集合定义比较特殊
字典(dict)	my_dict = {}	空字典用花括号定义

3.3.2 变量数据类型分类

在 3.3.1 节中读者已经熟悉了 Python 常用变量的定义,接下来笔者将对变量的数据类型简单地进行总结及分类。

Python 变量数据类型分为 6 种,包括 number(数字)、string(字符串)、tuple(元组)、list(列表)、set(集合)、dictionary(字典),其中 number 数据类型包含 int(整型)、float(浮点型)、bool(布尔型)、complex(复数)。这 6 种数据类型分为两类,即可变数据类型和不可变数据类型。不可变数据类型表示当此类型变量的数值发生变化时,变量内存地址也会改变;可变数据类型表示当此类型变量的数值发生变化时,变量内存地址不变。变量内存地址可以使用 id()方法获取。为了方便记忆,笔者将数据类型列出,见表 3-2。

表 3-2 Python 数据类型

数据类型分类	数 据 类 型	特　点
不可变数据类型	number、string、tuple	当不可变数据类型值变化时,内存地址也会变化
可变数据类型	list、set、dictionary	当可变数据类型值变化时,内存地址不会变化

1. 不可变数据类型

笔者以 int 变量为例演示不可变数据类型的特点,代码如下:

```
//第 3 章/new_python/my_python_2.py
x = 10
print("变量 x 的值为{}".format(x))
print("变量 x 的数据类型是:{}".format(type(x)))
print("变量 x 的内存地址是:{}".format(id(x)))
x = 20
print("变量 x 的值为{}".format(x))
print("变量 x 的数据类型是:{}".format(type(x)))
print("变量 x 的内存地址是:{}".format(id(x)))

#执行结果
变量 x 的值为 10
变量 x 的数据类型是:<class 'int'>
变量 x 的内存地址是:1535476112
变量 x 的值为 20
变量 x 的数据类型是:<class 'int'>
变量 x 的内存地址是:1535476432
```

示例中，笔者使用 id()方法获取了变量的内存地址，当 x 从 10 变为 20 时，内存地址从 1535476112 变为 1535476432，这就是不可变数据类型的特点。

2. 可变数据类型

笔者以 list 变量为例演示可变数据类型的特点，代码如下：

```
//第 3 章/new_python/my_python_3.py
my_list = ['栗子', '软件']
print("my_list 的值为{}".format(my_list))
print("my_list 的类型为{}".format(type(my_list)))
print("my_list 的内存地址是:{}".format(id(my_list)))
my_list.append('测试')
print("my_list 的值为{}".format(my_list))
print("my_list 的类型为{}".format(type(my_list)))
print("my_list 的内存地址是:{}".format(id(my_list)))

#执行结果
my_list 的值为['栗子', '软件']
my_list 的类型为<class 'list'>
my_list 的内存地址是:2761082241672
my_list 的值为['栗子', '软件', '测试']
my_list 的类型为<class 'list'>
my_list 的内存地址是:2761082241672
```

示例中，笔者使用列表的 append()方法给 my_list 列表增加了一个元素，从执行结果可以看出，当 my_list 列表从两个元素变为 3 个元素时，内存地址始终是 2096013083272，这就是可变数据类型的特点。

3.4 Python 运算符

在编码过程中，不可避免地会对多个变量进行运算，如四则运算、比较运算、逻辑运算等，本节中笔者将会演示 Python 如何使用常用的运算符。

3.4.1 算术运算符

算术运算符顾名思义是对变量进行加、减、乘、除等运算，读者需要了解的是变量如何进行算术运算和计算后如何赋值。

1. 加法

Python 使用“+”进行加法计算，加法表达式有两种写法，代码如下：

```
//第 3 章/new_python/my_python_4.py
#加法
x = 10
x = x + 1
print("x+1 的计算结果为{}".format(x))
y = 100
```

```
y += 1
print("y + 1 的计算结果为{}".format(y))

#执行结果
x + 1 的计算结果为 11
y + 1 的计算结果为 101
```

示例中,笔者采用了两种方式进行加法计算,一种是 x=x+1;另一种是 y+=1,从执行结果可以看出,这两种方式都可以实现加 1 计算。

2. 减法

Python 中减法计算也非常简单,只需使用“-”进行计算,代码如下:

```
//第 3 章/new_python/my_python_4.py
#减法
x = 10
x = x - 1
print("x - 1 的计算结果为{}".format(x))
y = 100
y -= 1
print("y - 1 的计算结果为{}".format(y))

#执行结果
x - 1 的计算结果为 9
y - 1 的计算结果为 99
```

示例中,笔者使用两种方式进行减法计算,和加法一样,两种方式都可以实现减 1 计算。

3. 乘法

Python 中乘法计算需要使用“*”进行计算,跟平常手写的乘法符号不一样,代码如下:

```
//第 3 章/new_python/my_python_4.py
#乘法
x = 10
x = x * 2
print("x * 2 的计算结果为{}".format(x))
y = 100
y *= 2
print("y * 2 的计算结果为{}".format(y))

#执行结果
x * 2 的计算结果为 20
y * 2 的计算结果为 200
```

示例中,乘法和加减法一样,也可以使用两种方式编写。

4. 除法

Python 的除法划分较细,一般分为 3 种情况。第 1 种除法无论是否整除结果均会保留小数;第 2 种除法会将结果向下取整;第 3 种除法会取除法的余数,代码如下:

```
//第 3 章/new_python/my_python_4.py
#除法
y = 100
y /= 2
print("y/2 的计算结果为{}".format(y))
#除法:向下整数
y = 99
y //= 2
print("y//2 的计算结果为{}".format(y))
#除法:取除法的余数
y = 99
y %= 2
print("y%2 的计算结果为{}".format(y))

#执行结果
y/2 的计算结果为 50.0
y//2 的计算结果为 49
y%2 的计算结果为 1
```

示例中,第 1 种除法使用“/”表示,100 除以 2 本应等于 50,但 Python 会保留小数点后一位,所以计算结果是 50.0;第 2 种除法使用“//”表示,99 除以 2 本应等于 49.5,但 Python 进行了向下取整,所以计算结果等于 49;第 3 种除法使用“%”表示,99 除以 2 本应等于 49 余数是 1,Python 直接取余数作为结果,所以计算结果等于 1。

5. 幂运算

Python 中使用“**”进行 x 的 n 次幂计算,代码如下:

```
//第 3 章/new_python/my_python_4.py
#x 的 n 次幂
x = 2
x **= 3
print("x 的 3 次幂, 计算结果为{}".format(x))

#执行结果
x 的 3 次幂, 计算结果为 8
```

示例中,笔者只使用了一种方式进行幂计算,读者可以根据前面的学习内容使用另外一种方式自行实验。

6. 总结

为了方便记忆,笔者对 Python 算术运算符的内容进行了总结,见表 3-3。

表 3-3 Python 算术运算符

算术运算符	描　　述	应　　用
+	加法	x += 1 或 x = x+1
−	减法	x −= 1 或 x = x−1
*	乘法	x *= 2 或 x = x*2

续表

算术运算符	描　　述	应　　用
/	除法	x /= 2 或 x = x/2
%	取除法的余数	x %= 2 或 x = x%2
//	除法向下整数	x //= 2 或 x = x//2
**	x 的 n 次幂	x **= 2 或 x = x** 2

3.4.2 比较运算符

比较运算符也是很容易理解的，就是对两个变量进行比较。Python 比较运算符有 6 种，分别是等于(==)、不等于(!=)、大于(>)、小于(<)、大于或等于(>=)、小于或等于(<=)，见表 3-4。

表 3-4 Python 比较运算符

比较运算符	描　　述	应　　用
==	等于	x==y
!=	不等于	x!=y
>	大于	x>y
>=	大于或等于	x>=y
<	小于	x<y
<=	小于或等于	x<=y

比较运算符一般会与分支语句一起使用，例如，如果比较结果满足条件，则执行代码 A，如果不满足条件，则执行代码 B。以等于为例，当两个变量值相等时结果会返回值 True，当两个变量值不相等时结果会返回值 False。接下来笔者将以等于和不等于为例，结合分支语句给读者简单演示比较运算符的用法，代码如下：

```
//第 3 章/new_python/my_python_5.py
#等于、不等于
x = 10
y = 100
if x != y:
    print(x != y)
    print('执行结果:x 不等于 y')

#执行结果
True
执行结果:x 不等于 y
```

示例中，笔者使用分支语句结合比较运算符进行了演示，虽然读者暂时还没有学习分支语句的知识，但从单词意思上也可以理解，if 语句的意思是如果条件成立，则打印比较结果和执行结果说明。

由于 x 等于 10，y 等于 100，所以两者一定是不相等的，执行结果中 x 和 y 不相等打印的是 True，表示 x 和 y 不相等。

3.4.3 逻辑运算符

Python 逻辑运算符有 3 种，分别是与(and)、或(or)、非(not)，and 表示两个条件都必须满足，即两个条件都为 True；or 表示两个条件只要满足一个即可；not 表示取反，即当条件为 True 时加 not，则条件变为 False，见表 3-5。

表 3-5 Python 逻辑运算符

逻辑运算符	描 述	应 用
and	与	x and y
or	或	x or y
not	非	not x

逻辑运算符可以在分支语句中联合多个条件进行判断，可以构造在多个条件都满足的情况下执行代码 A 的场景，也可以构造在满足其中一个条件的情况下执行代码 A 的场景，具体如何使用逻辑运算符还要看在实际开发过程中需要满足哪些条件。根据逻辑运算符的分析，笔者构造了一些简单的场景进行演示，代码如下：

```
//第 3 章/new_python/my_python_6.py
#and
x = 10
y = 100
if x>5 and y>50:
    print('and 执行结果:x 大于 5,并且 y 大于 50')
#or
x = 1
y = 100
if x>5 or y>50:
    print('or 执行结果:x 大于 5,或 y 大于 50')
#not
flag = False
if flag:
    print(flag)
if not flag:
    print(not flag)
    print('not 执行结果:flag 结果为 False')

#执行结果
and 执行结果:x 大于 5,并且 y 大于 50
or 执行结果:x 大于 5,或 y 大于 50
True
not 执行结果:flag 结果为 False
```

在 and 运算符示例中，x 等于 10，y 等于 100，满足 x 大于 5 且 y 大于 50，所以可以正常打印结果；在 or 运算符示例中，x 等于 1，y 等于 100，满足 x 大于 5 或 y 大于 50 中的一个条件，也可以正常打印结果；在 not 运算符示例中，变量 flag 的初始值是 False，那么 not flag 的值就是 True，所以打印的是 not flag 分支判断后的语句。

3.4.4 成员运算符

成员运算符用来判断变量 x 是否在 y 序列中,其中 y 序列可以是字符串、列表、元组等,in 表示在序列内,not in 表示不在序列内,见表 3-6。

表 3-6 Python 成员运算符

成员运算符	描　　述	应　　用
in	在序列中	x in y
not in	不在序列中	x not in y

笔者分别使用字符串、元组、列表演示成员运算符的使用,代码如下:

```
//第 3 章/new_python/my_python_7.py
#在字符串内
db_port = "3306"
db_str = "端口号是:3306"
if db_port in db_str:
    print('执行结果:{}在 db_str 字符串内'.format(db_port))
#在元组内
db_ip = '127.0.0.1'
db_tuple = ('127.0.0.1', 3306, 'admin', '123456')
if db_ip in db_tuple:
    print('执行结果:{}在 db_tuple 元组内'.format(db_ip))
#不在列表内
web_port = 8080
db_list = ['127.0.0.1', 3306, 'admin', '123456']
if web_port not in db_list:
    print('执行结果:{}不在 db_list 列表内'.format(web_port))

#执行结果
执行结果:3306 在 db_str 字符串内
执行结果:127.0.0.1 在 db_tuple 元组内
执行结果:8080 不在 db_list 列表内
```

成员运算符一般与分支语句一起使用,笔者对字符串和元组使用了 in 判断,对列表使用了 not in 判断,代码内容相对简单,读者可以自行练习并尝试理解。

3.4.5 身份运算符

身份运算比较的是对象的内存地址,is 表示内存地址相同,is not 表示内存地址不同,见表 3-7。

表 3-7 Python 身份运算符

身份运算符	描　　述	应　　用
is	内存地址相同	x is y
is not	内存地址不同	x is not y

笔者以字符串和列表为例，演示变量在值相同的情况下内存地址是否相同，代码如下：

```
//第 3 章/new_python/my_python_8.py
#字符串
db_ip = "127.0.0.1"
mysql_ip = "127.0.0.1"
if db_ip is mysql_ip:
    print('db_ip 和 mysql_ip 的内存地址相同')
if db_ip == mysql_ip:
    print('db_ip 和 mysql_ip 的值相同')
print(id(db_ip))
print(id(mysql_ip))
#列表
db_info = ['127.0.0.1', 3306, 'admin', '123456']
mysql_info = ['127.0.0.1', 3306, 'admin', '123456']
if db_info is not mysql_info:
    print('db_info 和 mysql_info 的内存地址不同')
if db_info == mysql_info:
    print('db_info 和 mysql_info 的值相同')
print(id(db_info))
print(id(mysql_info))

#执行结果
db_ip 和 mysql_ip 的内存地址相同
db_ip 和 mysql_ip 的值相同
2248746912304
2248746912304
db_info 和 mysql_info 的内存地址不同
db_info 和 mysql_info 的值相同
2248746762888
2248746912392
```

示例中，笔者使用 id()方法获取变量的内存地址，又使用了==来比较两个变量的值是否相等。得出的结论是，两个字符串变量的值相等、内存地址也相等；两个列表变量的值相等，但内存地址不相等。

3.5 Python 字符串

字符串在开发过程中是最常使用的，除了前面学到的简单的定义之外，字符串还有多种相关的操作，这些操作在实际工作中可以帮助读者解决很多问题，所以读者需要多多练习，以便在以后的工作中灵活应用。

3.5.1 字符串定义

首先简单复习一下 Python 中字符串的定义，使用单引号或双引号将内容引起来即可，代码如下：

```
//第 3 章/new_python/my_python_9.py
my_str1 = '栗子'
my_str2 = "测试"
print(type(my_str1))
print(type(my_str2))

#执行结果
<class 'str'>
<class 'str'>
```

在一般情况下两种定义字符串的方式并没有什么区别,但如果使用单引号定义字符串,同时字符串中包含英文的单引号,则使用单引号定义字符串就会报错,代码如下:

```
my_str1 = 'I'm 栗子'

#执行结果
SyntaxError: invalid syntax
```

示例中,定义字符串 my_str1 使用的是单引号,但单引号中还包含单引号。此时运行代码,Python 报错:SyntaxError: invalid syntax,表示语法错误。说明在单引号内不能包含多余的单引号。

如果想解决这个问题,读者则可以使用以下两种方式。方式一,使用单引号定义字符串,但在 I'm 中间的单引号前面加一个反斜杠,表示转义,即让 Python 把它当作一个单引号。方式二就更加简单,只需使用双引号定义包含单引号的字符串。这里比较推荐方式二,因为方法简单直接,代码如下:

```
//第 3 章/new_python/my_python_10.py
#转义
my_str1 = 'I\'m 栗子'
print(my_str1)
#使用双引号
my_str1 = "I'm 栗子"
print(my_str1)

#执行结果
I'm 栗子
I'm 栗子
```

3.5.2 字符串拼接

Python 字符串拼接通常使用两种方式。一种是使用加号进行拼接;另一种是在字符串中需要拼接内容处写{},然后使用 format()方法以参数形式按顺序填入相应值实现拼接。

在 Python 中使用加号进行拼接时,读者需要注意拼接变量的数据类型,如果多个字符串使用加号拼接,则 Python 可以正常处理,但如果对字符串和数字进行拼接,则 Python 会

报错,代码如下:

```
//第 3 章/new_python/my_python_11.py
#score 为字符串类型
name = "栗子"
str = "的分数是:"
score = "60"
result = name + str + score
print(result)
#score 为数字类型
name = "栗子"
str = " 的分数是:"
score = 60
result = name + str + score
print(result)

#执行结果
栗子的分数是:60
TypeError: must be str, not int
```

示例中,笔者先将 name、str、score 变量都定义为字符串,然后使用加号进行拼接并赋值给 result 变量,执行结果可以正常输出,但如果笔者将 score 变量赋值为数字,再使用加号进行拼接并赋值给 result 变量,则执行代码后 Python 会提示 TypeError: must be str, not int,意思是整型和字符串不能使用加号进行拼接。

如果想要对整型和字符串进行拼接,则可以使用 format()方法,代码如下:

```
//第 3 章/new_python/my_python_12.py
name = "栗子"
str = "的分数是:"
score = 60
result = "{}{}{}".format(name, str, score)
print(result)

#执行结果
栗子的分数是:60
```

示例中,result 变量在赋值时使用 3 个{}进行占位,然后使用 format()方法,此方法的参数依次为 name、str、score,表示拼接这 3 个变量。虽然 score 变量依然为整型,但执行结果并没有报错,这说明 format()方法可以对 int 类型和 str 类型的变量进行拼接,所以在变量拼接时推荐使用 format()方法。

3.5.3 字符串分割

在实际工作中,经常会遇到需要获取字符串中部分内容的情况。此时需要观察字符串的规律,然后使用 split()方法分割字符串,最终从字符串中得到需要的内容。例如想要从 3.5.2 节拼接好的字符串中提取分数,代码如下:

```
//第 3 章/new_python/my_python_13.py
score_info = "栗子的分数是:99"
score_info_list = score_info.split(":")
print(score_info_list)
print(score_info_list[1])

#执行结果
['栗子的分数是', '99']
99
```

示例中,score_info 变量值中分数和文字之间使用的是中文冒号,所以笔者使用 split()方法时传入参数中文冒号进行分割,分割结果赋值给变量 score_info_list。从执行结果可以看出,分割结果是一个列表,笔者只需打印列表下标为 1 的元素,即可得到分数值。

3.5.4 字符串替换

字符串替换在实际工作中的应用也比较广泛,在 Python 中使用 replace()方法指定被替换内容和替换内容后,即可简单地完成替换工作,代码如下:

```
//第 3 章/new_python/my_python_14.py
#字符串替换
my_url = "127.0.0.1:8000/v1/user/edit"
my_url2 = my_url.replace("/", " + ")
print(my_url2)

#执行结果
127.0.0.1:8000 + v1 + user + edit
```

示例中,笔者需要将 url 中的斜杠替换成加号,于是使用 replace()方法,replace()方法中的第 1 个参数为被替换内容,第 2 个参数为替换内容。从执行结果可以看出“/”全部被替换成“+”。

3.5.5 字符串删除前后空格

有时获取的字符串前面或后面会包含空格,如果直接使用包含前后空格的字符串,则会出现报错,这就需要想办法去掉字符串的前后空格,此时可以使用 strip()方法去掉字符串前后空格,代码如下:

```
//第 3 章/new_python/my_python_15.py
name = " 栗子 "
print(name)
name2 = name.strip()
print(name2)

#执行结果
  栗子
栗子
```

除了可以一次性去除左右空格外，也可以单独去除前边或后边的空格。去掉前面的空格可以使用 lstrip()方法；去掉后面的空格可以使用 rstrip()方法，代码如下：

```
//第 3 章/new_python/my_python_15.py
#去掉前面、后面的空格
name = " 栗子 "
print(name)
name2 = name.lstrip()
print(name2)
name3 = name.rstrip()
print(name3)

#执行结果
  栗子
栗子
  栗子
```

3.5.6 字符串大小写

在进行自动化测试时，如果让用户输入浏览器英文名决定使用哪个浏览器，读者就需要关注用户输入浏览器英文名大小写的问题了。用户的输入名字可能是大写的、可能是小写的、也可能是大小写混合的。此时要判断是哪个浏览器，就需要将用户输入内容转换为全部大写或全部小写进行比较，代码如下：

```
//第 3 章/new_python/my_python_16.py
#转换成大写
browser = "Firefox"
if browser.upper() == "FIREFOX":
    print(browser.upper())
    print("使用火狐浏览器")
#转换成小写
browser = "Chrome"
if browser.lower() == "chrome":
    print(browser.upper())
    print("使用谷歌浏览器")

#执行结果
FIREFOX
使用火狐浏览器
chrome
使用谷歌浏览器
```

示例中，upper()方法表示将所以字母转换为大写，lower()方法表示将所有字母转换为小写。在实际工作中，为了避免由于大小写不一致带来的问题，一般情况下先对变量进行统一转换，然后进行比较。


12min

3.6 Python 元组

元组是一种有序且不可更改的数据结构,即创建后不能对其进行修改,所以读者可以将一组不需要变化的数据保存在元组中。

3.6.1 元组定义

首先回顾一下元组的定义,主要关注的是空元组和只包含一个元素的元组,代码如下:

```
//第 3 章/new_python/my_python_18.py
#元组定义
my_tup1 = ()
my_tup2 = ("栗子",)
print(type(my_tup1))
print(type(my_tup2))

#执行结果
<class 'tuple'>
<class 'tuple'>
```

示例中,只包含一个元素的元组需要在元素后边加上逗号,如果不加逗号,Python 则会认为定义的是一个字符串,而不是元组。

3.6.2 元组访问

因为元组是有序的,所以读者可以使用下标来对其进行访问。元组的下标是从 0 开始的,所以在访问元组中的第 1 个元素时需要使用下标 0,代码如下:

```
#元组访问
my_tup = ('10.20.30.40', 8888, 'lizi', '123456')
print(my_tup[0])

#执行结果
10.20.30.40
```

3.6.3 元组遍历

元组的遍历一般使用两种方法,一种是使用 for 循环进行遍历;另一种是使用 for 循环结合 range()方法进行遍历。当然元组的遍历还有很多方法,但笔者在这里只介绍最常用的这两种方法。

1. for 循环遍历元组

for 循环遍历的格式是:for 变量 in 元组。在 for 循环的过程中,元组中的每个元素会被赋值给变量,这样读者就可以根据需求去使用这个变量了,代码如下:

```
//第 3 章/new_python/my_python_19.py
#元组遍历:for
db_info = ('127.0.0.1', 3306, 'admin', 123456)
for item in db_info:
    print(item)

#执行结果
127.0.0.1
3306
admin
123456
```

示例中,笔者使用 for 循环对数据库信息元组 db_info 进行了遍历,在遍历过程中将元组中的元素赋值给 item 变量,每循环一次打印一次 item 变量。从执行结果可以看出,for 循环共执行了 4 次,按顺序,每次循环打印元组中的一个值。

2. for 结合 range()方法遍历元组

此种遍历方法的格式是:for i in range(len(元组)),其中 range()方法用于根据所传参数生成一系列连续的整数,len()方法用于获取元组的长度。

1) range()方法含义

首先读者需要了解一下 range(len(元组))的含义,代码如下:

```
db_info = ('127.0.0.1', 3306, 'admin', 123456)
print(range(len(db_info)))

#执行结果
range(0, 4)
```

示例中,笔者直接打印了 range(len(db_info)),由于 len(db_info)方法得到元组的长度是 4,所以执行结果是 range(0, 4),表示返回 0 到 4 的整数。由于 range()方法也是左闭右开的,所以返回的整数是 0、1、2、3。

2) for 结合 range()方法遍历元组

熟悉了 range()方法后,读者就可以结合 for 循环来遍历元组了,代码如下:

```
//第 3 章/new_python/my_python_19.py
#元组遍历:for + range()
db_info = ('127.0.0.1', 3306, 'admin', 123456)
for i in range(len(db_info)):
    print(db_info[i])

#执行结果
127.0.0.1
3306
admin
123456
```

示例中,for 每次循环会把 range()方法返回的整数赋值给变量 i,笔者使用变量 i 充当

元组的下标,这样就可以达到遍历元组的目的了。从执行结果可以看出,笔者通过下标成功地打印了元组中所有的元素。

3.6.4 字符串切片

字符串切片简单理解就是提取字符串中的部分字符。要学会字符串切片,首先需要知道字符串中字符的下标。字符串正向下标从 0 开始,反向下标从−1 开始,如图 3-1 所示。

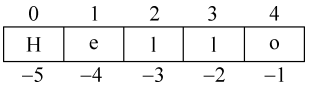

图 3-1 Python 字符串中字符下标

如果读者想获取字符串中的第 1 个位置的内容,则只需使用下标 0。如果读者想获取字符串中的某几个位置的内容,则需要写明开始位置下标和结束位置下标,并且两个下标中间用冒号分隔。例如 3.5.3 节获取学员分数的例子,使用字符串切片也可以实现,代码如下:

```
//第 3 章/new_python/my_python_17.py
#字符串切片:获取第 1 个字
score_info = "栗子的分数是:98"
first = score_info[0]
print(first)
#字符串切片:获取分数
score_info = "栗子的分数是:98"
score = score_info[7:9]
print(score)

#执行结果
栗
98
```

示例中,score_info[0]表示获取字符串中的第 1 个位置的内容; score_info[7:9]表示从左到右,获取字符串中第 7 个位置到第 8 个位置的内容。由于切片区间是左闭右开的,所以在获取第 7 个位置到第 8 个位置的内容时结束位置要写 9,不然无法获取第 8 个位置的内容。

当读者获取分数时,也可以从右到左进行反向获取,不过读者需要记住,反向的初始下标为−1。根据左闭右开的规则,在获取分数时右边不能写−1,代码如下:

```
//第 3 章/new_python/my_python_17.py
#字符串切片:反向获取
score_info = "栗子的分数是:98"
score = score_info[-2:]
print(score)

#执行结果
98
```

示例中,笔者并没有写结束位置内容,从执行结果来看,可以正确地获取想要的分数,说明写法并没有问题。那么如果结束位置写−1,则结果会如何呢? 读者可以参考上面的代码

自行练习，以便更好地掌握字符串切片。

3.7 Python 列表

列表是一种有序且可改变的数据结构。如果既想保存一组数据，又想在需要时改变其中的某个数据，就可以将数据保存在列表中。

3.7.1 列表定义及访问

同元组一样，读者需要首先关注空列表和只包含一个元素的列表，代码如下：

```
//第 3 章/new_python/my_python_20.py
#列表定义
my_list1 = []
my_list2 = ['栗子']
print(type(my_list1))
print(type(my_list2))

#执行结果
<class 'list'>
<class 'list'>
```

列表跟元组一样，也是有序的，所以读者可以继续使用下标对列表进行访问，代码如下：

```
#列表访问
db_info = ['127.0.0.1', 3306, 'admin', 123456]
print(db_info[0])

#执行结果
127.0.0.1
```

3.7.2 列表增、删、改操作

列表与元组的不同在于列表可以对其中的元素进行操作，包括增、删、改，所以列表在实际工作中应用得更加广泛，读者应该熟悉列表的基本操作。

1. 列表新增操作

列表新增元素的方式有两种，一种是在列表的末尾追加一个元素；另一种是在指定下标位置插入一个元素。

1）追加

在列表末尾追加元素使用 append()方法，将追加内容作为 append()方法的参数传入即可，代码如下：

```
#列表追加
db_info = ['127.0.0.1', 3306, 'admin', 123456]
```

```
db_info.append("栗子测试")
print(db_info)
#执行结果
['127.0.0.1', 3306, 'admin', 123456, '栗子测试']
```

示例中,笔者使用 append()方法在 db_info 列表的最后追加了一个字符串,从执行结果可以看出,通过追加操作将字符串追加到列表的最后。

2) 插入

在列表的指定位置插入元素使用 insert()方法。此方法需要传入两个参数,第 1 个参数表示要插入的位置;第 2 个参数表示要插入的元素内容,代码如下:

```
#列表插入
db_info = ['127.0.0.1', 3306, 'admin', 123456]
db_info.insert(0, "栗子测试")
print(db_info)

#执行结果
['栗子测试', '127.0.0.1', 3306, 'admin', 123456]
```

示例中,笔者使用 insert()方法在下标为 0 的位置插入了一个字符串,从执行结果可以看出,通过插入操作将字符串插入列表的第 1 个位置。

2. 列表修改操作

列表修改操作非常简单,只需找到需要修改的元素,然后对其重新赋值。例如想要修改数据库列表中的用户名,代码如下:

```
#列表修改
db_info = ['127.0.0.1', 3306, 'admin', 123456]
db_info[2] = 'lizi'
print(db_info)

#执行结果
['127.0.0.1', 3306, 'lizi', 123456]
```

示例中,笔者观察数据库列表中用户名的下标是 2,所以笔者对 my_list[2]重新进行赋值。从执行结果可以看出,通过重新赋值数据库列表,用户名已经被修改。

3. 列表删除操作

删除列表元素的方法比较多,如删除最后一个元素、删除指定下标的元素、根据元素值删除元素、清空列表等。

1) 删除最后一个元素

删除最后一个元素只需使用 pop()方法,pop()方法不需要传入任何参数,代码如下:

```
#列表修改
db_info = ['127.0.0.1', 3306, 'admin', 123456]
db_info.pop()
```

```
print(db_info)

#执行结果
['127.0.0.1', 3306, 'admin']
```

示例中，笔者调用 pop()方法，表示删除最后一个元素。从执行结果可以看出，db_info 列表中最后一个元素已经被删除。

2）删除指定下标的元素

删除指定下标的元素也可使用 pop()方法，只不过方法中需要传参数，参数内容为需要删除元素的下标，代码如下：

```
#删除指定下标的元素
db_info = ['127.0.0.1', 3306, 'admin', 123456]
db_info.pop(1)
print(db_info)

#执行结果
['127.0.0.1', 'admin', 123456]
```

示例中，笔者调用 pop()方法时传入的参数为 1，表示需要删除 db_info 列表中下标为 1 的元素，即删除端口号。从执行结果可以看出，db_info 列表中端口号已经被删除。

3）删除指定元素

当删除指定元素时需要使用 remove()方法，该方法的参数是需要删除的元素值，代码如下：

```
#删除指定元素
db_info = ['127.0.0.1', 3306, 'admin', 123456]
db_info.remove(123456)
print(db_info)

#执行结果
['127.0.0.1', 3306, 'admin']
```

示例中，笔者调用 remove()方法且传入参数为密码，从执行结果可以看出，db_info 列表中密码已经被删除。

3.7.3 列表遍历

列表遍历方式也有很多种，笔者在这里只介绍两种最常用的方式。一种是使用 for 循环进行遍历；另一种是使用 for 循环结合 range()方法进行遍历。这两种遍历方式和元组中介绍的遍历方式相同，相信读者一看便能了解其中的含义。

1. for 循环遍历列表

使用 for 循环遍历时，每次循环都会将元素赋值给变量 item，读者可以按需使用 item 变量，代码如下：

```
#列表遍历:for
db_info = ['127.0.0.1', 3306, 'admin', 123456]
for item in db_info:
    print(item)

#执行结果
127.0.0.1
3306
admin
123456
```

2. for 结合 range()方法遍历列表

使用 range()方法生成一系列整数,并将整数赋值给循环变量 i,读者可以根据下标 i 使用列表元素,代码如下:

```
//第 3 章/new_python/my_python_24.py
#列表遍历:for + range()
db_info = ['127.0.0.1', 3306, 'admin', 123456]
for i in range(len(db_info)):
    print(db_info[i])

#执行结果
127.0.0.1
3306
admin
123456
```


11min

3.8 Python 集合

集合是一种无序且不可重复的数据结构,所以读者可以将一组不需要排序的数据放在集合中。

3.8.1 集合定义及访问

空集合需要使用 set()来定义,将非空集合数据写在花括号内即可,代码如下:

```
//第 3 章/new_python/my_python_25.py
#集合定义
my_set1 = set()
my_set2 = {'栗子'}
print(type(my_set1))
print(type(my_set2))

#执行结果
```

```
<class 'set'>
<class 'set'>
```

集合还有一个不可重复的特性，即当集合中有相同的元素时，只保留一个。例如数据库信息集合中包含两个端口号 3306，那么打印时只会保留一个，代码如下：

```
#集合不可重复
db_info = {'127.0.0.1', 3306, 'admin', 123456, 3306}
print(db_info)

#执行结果
{123456, 3306, 'admin', '127.0.0.1'}
```

另外，由于集合是无序的，所以不能用下标访问，如果使用下标访问，则会报错，代码如下：

```
#使用集合下标访问
db_info = {'127.0.0.1', 3306, 'admin', 123456}
print(db_info[0])

#执行结果
TypeError: 'set' object does not support indexing
```

3.8.2 集合应用

集合在实际工作中应用得比较少，应用集合的场景也不是对集合进行增、删遍历，而是使用集合进行去重、获取两个集合的交集、获取两个集合的并集等，所以笔者在本节只会简单地介绍集合的实际应用，不再介绍集合的增、删遍历操作。

1. 集合去重

例如需要将列表中的数据去重，在不考虑列表顺序的情况下，可以先将列表转换成集合，然后将集合转换成列表，即可达到去重的目的，代码如下：

```
//第 3 章/new_python/my_python_26.py
#列表去重
db_info = ['127.0.0.1', 3306, 'admin', 123456, 123456, 123456]
db_info_set = set(db_info)
print(type(db_info_set))
print(db_info_set)
db_info_list = list(db_info_set)
print(type(db_info_list))
print(db_info_list)

#执行结果
<class 'set'>
{123456, 3306, '127.0.0.1', 'admin'}
<class 'list'>
[123456, 3306, '127.0.0.1', 'admin']
```

示例中,笔者想要对 db_info 列表数据进行去重操作。笔者先使用 set()方法将列表转换成集合,从执行结果可以看出,转换成集合后重复内容会被自动删除,然后笔者使用 list()方法再将集合转换成列表,从执行结果可以看出,集合可以被正常转换成列表且元素已经去重,但列表中的元素顺序发生了改变,所以读者在使用此方法进行去重时一定要考虑列表顺序是否允许打乱。

2. 集合交集

求两个集合的交集需要使用 intersection()方法,格式为集合 A.intersection(集合 B)。通过获取交集操作可以得一个新的集合,集合中的元素是集合 A 和集合 B 中都存在的元素,代码如下:

```
//第 3 章/new_python/my_python_27.py
#集合交集
my_set1 = {'10.20.30.40', 8888, 'lizi', '123456'}
my_set2 = {'192.168.0.100', 3306, 'lizi', '123456'}
print(my_set1.intersection(my_set2))

#执行结果
{'lizi', '123456'}
```

示例中,my_set1 和 my_set2 相同的内容为用户名和密码元素,笔者使用 intersection()方法获取两个集合的交集。从执行结果可以看出,获取交集得到的新集合为用户名和密码的集合。

3. 集合并集

求两个集合的并集需要使用 union()方法,格式为集合 A.union(集合 B)。通过获取并集操作可以得到一个新的集合,集合中的元素是两个集合的所有元素,其中重复元素只保留一个,代码如下:

```
//第 3 章/new_python/my_python_27.py
#集合并集
db_info = {'127.0.0.1', 3306, 'admin', 123456}
web_info = {'https://www.lizi.com', 8080, 'admin', 123456}
result = db_info.union(web_info)
print(type(result))
print(result)

#执行结果
<class 'set'>
{123456, 3306, '127.0.0.1', 'admin', 8080, 'https://www.lizi.com'}
```

示例中,db_info 和 web_info 相同的内容为用户名和密码元素,笔者使用 union()方法获取两个集合的并集。从执行结果可以看出,获取并集得到的新集合包含两个集合的所有元素,并且对两个集合中的相同元素进行了去重操作。

4. 集合差集

求两个集合的差集需要使用 difference()方法,格式为集合 A.difference(集合 B),返回

的是集合 A 中与集合 B 不相同的元素的集合，代码如下：

```
//第 3 章/new_python/my_python_27.py
#集合差集
db_info = {'127.0.0.1', 3306, 'admin', 123456}
web_info = {'https://www.lizi.com', 8080, 'admin', 123456}
result = db_info.difference(web_info)
print(type(result))
print(result)
result = web_info.difference(db_info)
print(type(result))
print(result)

#执行结果
<class 'set'>
{'127.0.0.1', 3306}
<class 'set'>
{8080, 'https://www.lizi.com'}
```

示例中，db_info.difference(web_info)返回的是 db_info 与 web_info 不同的元素的集合，web_info.difference(db_info)返回的是 web_info 与 db_info 不同的元素的集合。

3.8.3 元组列表集合的区别

学了元组、列表、集合这 3 种类似的数据结构后，笔者对这 3 种数据结构的特点进行了总结，以便读者有效地进行记忆，见表 3-8。

表 3-8 Python 元组、列表、集合区别

数据结构	特　点	下标访问
元组	有序、可重复、只读	可以
列表	有序、可重复、读写	可以
集合	无序、不可重复、读写	不可以

3.9 Python 字典


11min

字典是一种可变容器，可以存储任意类型的数据。字典的格式为{key：value}，其中 key 是不可以重复的。

3.9.1 字典定义

当定义空字典时，可以直接使用空的花括号；当定义非空字典时，字典中的数据需要使用 key:value 格式，代码如下：

```
//第 3 章/new_python/my_python_28.py
#空字典
db_info = {}
```

```
print(type(db_info))
#非空字典
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456}
print(type(db_info))
print(db_info)

#执行结果
<class 'dict'>
<class 'dict'>
{'host': '127.0.0.1', 'port': 3306, 'username': 'admin', 'password': 123456}
```

当字典中的 key 重复时 Python 并不会报错,而是只保存字典中最后一个键-值对,忽略前面的重复内容,代码如下:

```
//第 3 章/new_python/my_python_28.py
#key 重复
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456, 'host':
'https://www.lizi.com'}
print(db_info)

#执行结果
{'host': 'https://www.lizi.com', 'port': 3306, 'username': 'admin', 'password': 123456}
```

示例中,笔者在字典中添加了两个同样的键 host,但两个 host 赋值不同,目的是分辨当键相同时字典保存的是哪一个。从执行结果可以看出,字典保存的是后面的 host。

3.9.2 字典访问

由于字典中元素是以 key:value 格式存放的,所以直接使用 key 进行访问即可得到对应的 value 值,而不需要考虑顺序问题,代码如下:

```
//第 3 章/new_python/my_python_29.py
#字典访问
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456}
print(db_info['username'])

#执行结果
admin
```

除了可以直接使用 key 进行访问外,还可以通过 get()方法传入参数 key 进行访问,此方法不常用,读者只需了解,代码如下:

```
//第 3 章/new_python/my_python_29.py
#get()方法访问
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456}
print(db_info.get('username'))

#执行结果
admin
```

3.9.3 字典增、删、改操作

字典是 key:value 结构的,所以字典的增、删、改都可以围绕 key 进行。

1. 字典新增操作

字典新增只需新增 1 个 key,并且给这个 key 赋值,格式为字典[key]=value,代码如下:

```
//第 3 章/new_python/my_python_30.py
#字典新增
db_info = {'host':'127.0.0.1'}
db_info['port'] = 3306
print(db_info)

#执行结果
{'host': '127.0.0.1', 'port': 3306}
```

示例中,笔者新增键 port 并赋值为 3306,从执行结果可以看出,字典新增 port 成功了。

2. 字典修改操作

修改字典需要先确认修改数据的 key 是什么,然后对该 key 重新赋值即可,代码如下:

```
//第 3 章/new_python/my_python_30.py
#字典修改
db_info = {'host':'127.0.0.1'}
db_info['host'] = 'https://www.lizi.com'
print(db_info)

#执行结果
{'host': 'https://www.lizi.com'}
```

示例中,笔者对 host 进行了重新赋值,从执行结果可以看出,字典中 host 值发生了相应改变。

3. 字典删除操作

删除操作也需要先确认删除数据的 key 是什么,然后使用 pop()方法将参数传入 key 即可,代码如下:

```
//第 3 章/new_python/my_python_30.py
#字典删除
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456}
db_info.pop('host')
print(db_info)

#执行结果
{'port': 3306, 'username': 'admin', 'password': 123456}
```

示例中,笔者调用 pop()方法且传入的 key 为 host,从执行结果可以看出,字典中的 host 被删除。

3.9.4 字典遍历

字典遍历比较特殊,读者可以单独遍历字典的 key,也可以单独遍历字典的 value,还可以一起遍历字典的 key 和 value。

1. 遍历字典 key

遍历字典 key 时需要使用字典的 keys()方法获取字典所有键组成的可迭代对象,然后使用 for…in…语句进行遍历,代码如下:

```
//第 3 章/new_python/my_python_31.py
#遍历字典 key
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456}
for key in db_info.keys():
    print(key)

#执行结果
host
port
username
password
```

示例中,笔者调用 keys()方法获取 db_info 列表所有的 key 值,并在每次循环时进行打印。从执行结果可以看出,db_info 列表的 key 值全部正确地被打印了。

2. 遍历字典 value

同遍历字典 key 一样,遍历字典 value 时需要先使用字典的 values()方法获取字典的所有值组成的可迭代对象,然后使用 for…in…语句进行遍历,代码如下:

```
//第 3 章/new_python/my_python_31.py
#遍历字典 value
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456}
for value in db_info.values():
    print(value)

#执行结果
127.0.0.1
3306
admin
123456
```

示例中,笔者调用 values()方法获取 db_info 列表所有的 value 值,并在每次循环时进行打印。从执行结果可以看出,db_info 列表的 value 值全部正确地被打印了。

3. 遍历字典 key 和 value

当遍历字典的 key 和 value 时,需要先使用字典的 items()方法获取字典的所有 key 和 value 组成的可迭代对象,然后使用 for…in…语句进行遍历,代码如下:

```
//第 3 章/new_python/my_python_31.py
#遍历字典的 key 和 value
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456}
for item in db_info.items():
    print(item)

#执行结果
('host', '127.0.0.1')
('port', 3306)
('username', 'admin')
('password', 123456)
```

示例中,遍历时将字典的 key 和 value 当作了一个整体,即 item,从执行结果可以看出,每次循环 item 的值时获取的是 key 和 value 组成的元组。

读者也可以每次遍历时分别获取 key 和 value 的值,代码如下:

```
//第 3 章/new_python/my_python_31.py
#遍历字典 key 和 value
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456}
for key,value in db_info.items():
    print("key 是:{};value 是:{}".format(key, value))

#执行结果
key 是:host;value 是:127.0.0.1
key 是:port;value 是:3306
key 是:username;value 是:admin
key 是:password;value 是:123456
```

示例中,笔者将 for 循环中的 item 改为 key,value,这样在每次循环时可以先单独获取 key 或 value 的值,然后进行使用。从执行结果可以看出,获取的内容不再是元组。

3.10 Python 分支和循环

分支和循环在测试开发过程中是必不可少的,在实际工作中需要通过分支来判断应该执行哪些语句,通过循环来多次执行一些语句。前面内容中读者已经见过一些分支和循环的用法,本节中笔者将会对分支和循环的用法进行细化。

3.10.1 分支

分支很简单,即如果条件为 True,则执行语句 A,如果条件为 False,则执行语句 B。接下来笔者以计算学生的成绩为例,使用分支语句按照不同的分数打印不同的成绩。

在演示之前,笔者先简单地定义分数和成绩的关系,见表 3-9。

表 3-9 大学生成绩表

分　　数	成　　绩
90～100	优
80～89	良
60～79	中
60 以下	差

1. 单分支结构

单分支结构只包含一个 if 关键字。当条件成立时执行 if 关键字下面的代码，代码如下：

```
#单分支结构
score = 90
if 100 >= score >= 90:
    print('成绩:优')

#执行结果
成绩:优
```

示例中，笔者使用分支语句判断分数是否大于或等于 90、小于或等于 100，如果满足条件，则成绩为优。由于笔者将分数定义为 90，所以执行结果为优。

2. 双分支结构

双分支结构包含一个 if 关键字和一个 else 关键字。当条件成立时执行 if 关键字下的代码；当条件不成立时执行 else 关键字下的代码，代码如下：

```
//第 3 章/new_python/my_python_32.py
#双分支结构
score = 80
if 100 >= score >= 90:
    print('成绩:优')
else:
    print('成绩:不是优')

#执行结果
成绩:不是优
```

示例中，笔者假设分数大于或等于 90、小于或等于 100 成绩为优，否则成绩不是优，所以在 if 关键字下的语句中打印优，在 else 关键字下的语句中打印不是优。由于笔者此次将分数定义为 80，所以执行结果为不是优。

3. 多分支结构

多分支结构包含 if 关键字、elif 关键字和 else 关键字，其中 elif 关键字可以包含多个，用于处理多个条件判断，代码如下：

```
//第 3 章/new_python/my_python_32.py
#多分支结构
```

```
score = 70
if 100 >= score >= 90:
    print('成绩:优')
elif 90 > score >= 80:
    print('成绩:良')
elif 80 > score >= 60:
    print('成绩:中')
elif 60 > score >= 0:
    print('成绩:差')
else:
    print('成绩输入不正确')

#执行结果
成绩:中
```

示例中,笔者使用 if 关键和 elif 关键字进行了多次成绩判断,最后还使用 else 关键字进行错误提示。由于笔者此次将分数定义为 70,符合成绩大于或等于 60、小于 80 的判断,所以执行结果为中。

4. 嵌套分支结构

嵌套分支结构是前面提到的分支结构的综合应用,代码如下:

```
//第 3 章/new_python/my_python_32.py
#嵌套分支结构
score = 'a'
if isinstance(score, int) and 100 >= score >= 0:
    if 100 >= score >= 90:
        print('成绩:优')
    elif 90 > score >= 80:
        print('成绩:良')
    elif 80 > score >= 60:
        print('成绩:中')
    elif 60 > score >= 0:
        print('成绩:差')
else:
    print('请输入 0～100 的整数!')

#执行结果
请输入 0～100 的整数!
```

示例中,笔者使用了 isinstance()方法,该方法的作用是判断一个对象是不是想要的数据类型。该方法需要传两个参数,第 1 个参数是用户输入的对象;第 2 个参数是期望的数据类型。那么 isinstance(score, int)就表示判断用户输入的 score 变量是不是 int 类型,如果是 int 类型,则返回值为 True,如果不是 int 类型,则返回值为 False。

理解了 isinstance()方法后,示例中嵌套分支结构的意义就变得比较清晰了。外层分支判断用户输入的分数是不是整数,并且值在 0～100,如果用户输入正确,则进行内层分数判断,最终输出成绩;如果用户输入错误,则直接提示用户需要输入 0～100 的整数。


6min

3.10.2 循环

Python 中有两种循环方式,一种是 for 循环,另一种是 while 循环。前面的小节中笔者已经使用 for 循环对元组、列表、字典进行了遍历,本节中还会介绍如何跳出 for 循环和继续 for 循环的操作,然后使用 while 循环实现与 for 循环同样的功能。

1. for 循环

for 循环的格式为 for 迭代变量 in 迭代对象,其中迭代对象可以是有序列表,也可以是 range()方法定义的范围;迭代变量在循环过程中根据迭代对象的范围发生变化。

1) 基本用法

这里笔者还是以遍历字典为例带读者复习 for 循环的基本用法,代码如下:

```
//第 3 章/new_python/my_python_33.py
#遍历字典
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456}
for key,value in db_info.items():
    print("key 是:{};value 是:{}".format(key, value))

#执行结果
key 是:host;value 是:127.0.0.1
key 是:port;value 是:3306
key 是:username;value 是:admin
key 是:password;value 是:123456
```

2) continue 命令

continue 命令的作用是跳过此次循环,进入下一次循环。例如当遍历字典时如果 key 等于 host,则进入下一次循环,如果 key 不等于 host,则打印 key 和 value,此时就需要使用 continue 命令,代码如下:

```
//第 3 章/new_python/my_python_33.py
#continue
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456}
for key,value in db_info.items():
    if key == "host":
        continue
    print("key 是:{};value 是:{}".format(key, value))

#执行结果
key 是:port;value 是:3306
key 是:username;value 是:admin
key 是:password;value 是:123456
```

示例中,笔者在循环语句中使用分支语句判断 key 是否等于 host,如果等于,则调用 continue 命令进入下一次循环,如果不等于,则打印 key 和 value 的值。

3) break 命令

break 命令的作用是结束循环,不再遍历后边的内容。例如在遍历字典时如果 key 等

于 username，则结束循环，此时需要使用 break 命令实现，代码如下：

```
//第 3 章/new_python/my_python_33.py
#break
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456}
for key,value in db_info.items():
    if key == "username":
        break
    print("key 是:{};value 是:{}".format(key, value))

#执行结果
key 是:host;value 是:127.0.0.1
key 是:port;value 是:3306
```

示例中，笔者在循环语句中使用分支语句判断 key 是否等于 username，如果等于，则调用 break 语句终止循环，如果不等于，则打印 key 和 value 的值。

4）嵌套循环

在实际工作中有时需要两个 for 循环进行嵌套使用，例如 Excel 表格中有 3 行数据，每行有两个值，获取每个值并进行打印，代码如下：

```
//第 3 章/new_python/my_python_33.py
#嵌套循环
for i in range(1, 4):
    for j in range(1, 3):
        print('第{}行, 第{}个数据'.format(i, j))

#执行结果
第 1 行, 第 1 个数据
第 1 行, 第 2 个数据
第 2 行, 第 1 个数据
第 2 行, 第 2 个数据
第 3 行, 第 1 个数据
第 3 行, 第 2 个数据
```

示例中，外层循环执行一次，内层循环会执行两次，这就是嵌套循环的执行过程。读者需要多加练习，以便理解嵌套循环的用法和执行过程。

5）总结

一般情况下会使用 for 循环进行各种遍历，并且每种遍历方式略有不同，尤其是字典的遍历更为特殊，为了方便记忆，笔者做了总结，见表 3-10。

表 3-10 for 循环总结

循 环 要 求	循 环 代 码
循环 5 次	for i in range(0, 5)
遍历元组	for item in my_tup
结合 range()方法遍历元组	for i in range(0, len(my_tup))
遍历列表	for item in my_list

续表

循 环 要 求	循 环 代 码
结合 range()方法遍历列表	for i in range(len(my_list))
遍历字典 key	for key in my_dict.keys()
遍历字典 value	for value in my_dict.values()
遍历字典 key 和 value	for key, value in my_dict.items()
遍历字典中的每个元素	for item in my_dict.items()

2. while 循环

while 循环的格式为 while 条件。当条件为真时执行循环,直到条件不满足时停止循环。本节中仅介绍 while 循环遍历列表和字典,在实际工作中还是 for 循环使用得比较多。

1) 遍历列表

当使用 while 循环时需要关注两个问题,一个是循环几次;另一个是循环时如何获取列表数据,代码如下:

```
//第 3 章/new_python/my_python_34.py
#while 循环遍历列表
db_info = ['127.0.0.1', 3306, 'admin', 123456]
i = 0
while i < len(db_info):
    print(db_info[i])
    i = i + 1

#执行结果
127.0.0.1
3306
admin
123456
```

示例中,笔者在 while 循环的外边定义了一个 i 变量作为循环变量,当 i 小于列表长度时执行循环,每次循环后对 i 进行加 1 操作,确保循环次数正常、循环取值正常。如果读者不进行 i 加 1 操作,则 i 一直等于 0,并且一直小于列表长度,即循环条件一直为 True,这样就会出现死循环现象,结果会一直打印列表的第 1 个值。当多次循环后,i 的值不再小于列表长度时,while 循环就结束了。

2) 遍历字典

由于字典是 key:value 格式,所以为了确保在循环过程中可以使用 key 获取对应的 value 值,首先需要考虑获得字典中的每个 key,代码如下:

```
//第 3 章/new_python/my_python_34.py
#while 循环遍历字典
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456}
keys = list(db_info.keys())
i = 0
while i < len(keys):
```

```
        key = keys[i]
        print("key 是:{};value 是:{}".format(key, db_info[key]))
        i = i+1

    #执行结果
    key 是:host;value 是:127.0.0.1
    key 是:port;value 是:3306
    key 是:username;value 是:admin
    key 是:password;value 是:123456
```

示例中,笔者先使用字典 keys()方法获得字典中所有的 key,然后使用 list()方法得到 key 的列表,目的是通过列表有序的特性进行循环。循环过程中笔者使用下标获取 keys 列表中的每个 key,再通过 key 获取对应的 value 值,这样就达到了获取 key 和 value 的目的。

3) continue 和 break 命令

while 循环中继续循环和终止循环的命令和 for 循环中的一样,即都使用 continue 和 break 命令,代码如下:

```
//第 3 章/new_python/my_python_34.py
#continue
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456}
keys = list(db_info.keys())
i = 0
while i < len(keys):
    key = keys[i]
    i = i + 1
    if key == 'host':
        print("continue")
        continue
    print("key 是:{};value 是:{}".format(key, db_info[key]))

#执行结果
continue
key 是:port;value 是:3306
key 是:username;value 是:admin
key 是:password;value 是:123456
```

示例中,笔者在通过 i 获取 key 后,就进行了 i 加 1 操作,目的是无论 Python 执行跳出循环代码,还是执行打印 key 和 value 代码都可以做到对 i 进行加 1 操作。当然读者可以先在 continue 命令上方进行 i 加 1 操作,然后在打印下方进行 i 加 1 操作,这样也可以达到目的。如果是初学阶段,则读者完全不用考虑如何实现得更好的问题,代码能达到目的即可,随着代码学习、应用的深入读者写的代码自然会变得更合理。

break 命令的代码跟 continue 命令代码类似,这里笔者只进行演示,不再进行详细讲解,代码如下:

```
//第 3 章/new_python/my_python_34.py
#break
```

```
db_info = {'host':'127.0.0.1', 'port':3306, 'username':'admin', 'password':123456}
keys = list(db_info.keys())
i = 0
while i < len(keys):
    key = keys[i]
    i = i + 1
    if key == 'username':
        print("break")
        break
    print("key 是:{};value 是:{}".format(key, db_info[key]))

#执行结果
key 是:host;value 是:127.0.0.1
key 是:port;value 是:3306
break
```

3.10.3 分支循环综合应用

冒泡排序就是一个很好的分支循环综合应用的例子,而且在进行招聘面试时也是一个会经常考的算法题。例如有一个列表 my_list = [5, 4, 2, 3, 1],需要使用冒泡排序算法对列表元素以从小到大的顺序进行排序。

冒泡排序的做法是先用列表中的第 1 个值与第 2 个值进行对比,如果第 1 个值大于第 2 个值,则第 1 个值和第 2 个值调换位置,否则不进行任何操作,然后用第 2 个值与第 3 个值进行比较,如果第 2 个值大于第 3 个值,则第 2 个值和第 3 个值调换位置,否则不进行任何操作,以此类推。所有元素都比较完一轮后,列表的最后一个元素就是值最大的元素了,即已经排好序了,然后重复上述比较操作,但每次重复不需要再比较已经排好序的元素,直到没有需要比较的元素为止,代码如下:

```
//第 3 章/new_python/my_python_35.py
#冒泡排序
my_list = [5, 4, 3, 2, 1]
for i in range(len(my_list) - 1):
    print(i)
    for j in range(len(my_list) - 1 - i):
        if my_list[j]> my_list[j + 1]:
            my_list[j], my_list[j + 1] = my_list[j + 1], my_list[j]
        print("i = {}; j = {}; my_list = {}".format(i, j, my_list))

#执行结果
0
i = 0; j = 0; my_list = [4, 5, 3, 2, 1]
i = 0; j = 1; my_list = [4, 3, 5, 2, 1]
i = 0; j = 2; my_list = [4, 3, 2, 5, 1]
i = 0; j = 3; my_list = [4, 3, 2, 1, 5]
1
```

```
i = 1; j = 0; my_list = [3, 4, 2, 1, 5]
i = 1; j = 1; my_list = [3, 2, 4, 1, 5]
i = 1; j = 2; my_list = [3, 2, 1, 4, 5]
2
i = 2; j = 0; my_list = [2, 3, 1, 4, 5]
i = 2; j = 1; my_list = [2, 1, 3, 4, 5]
3
i = 3; j = 0; my_list = [1, 2, 3, 4, 5]
```

示例中，外层 for 循环中 range(len(my_list)－1)表示循环次数为列表长度减 1，内层 for 循环的比较次数需要根据外层循环确定，因为内层 for 循环执行完最大数就会放在列表的最后，这样以后就不需要对其进行比较了，所以内层 for 循环的范围是 range(len(my_list)－1－i)。

当前一个数大于后一个数时，笔者对两个数据的位置进行调换，即 my_list[j], my_list[j+1] = my_list[j+1], my_list[j]，这是 Python 定义、赋值两个变量的一种写法，可以对两个数赋值进行调换。最后从执行结果可以看出，经过冒泡排序后，列表最后按照从小到大的顺序排序成功。

3.11 Python 方法

前面学习的代码都是在一个文件中按照顺序编写并执行的，如果读者想在其他文件中再次使用这些代码，则需要将代码复制粘贴一份才可以使用。在实际工作中经常会重复使用某些代码，此时就需要把代码封装成方法，在使用时直接调用该方法即可。

3.11.1 Python 方法简介

Python 中方法的定义很简单，格式为 def 方法名()。如果方法需要接收用户输入，则可以给方法添加参数；如果方法需要将结果返回给用户，则可以在方法体内添加 return 进行返回。例如要求编写一个加法方法，用户可以输入两个数，最终得到两个数相加的结果，代码如下：

```
//第 3 章/new_python/test_add.py
#加法方法
def my_add(x, y):
    return x + y

if __name__ == "__main__":
    add_result = my_add(2, 3)
    print("加法计算的结果是:{}".format(add_result))

#执行结果
加法计算的结果是:5
```

示例中,笔者定义了加法 my_add()方法,用户在使用时只需输入 x 和 y 即可得到加法计算结果,其中 x 和 y 是方法 my_add()的参数,return x+y 表示此方法返回 x+y。笔者调用 my_add()方法,传入参数 2 和 3 表示想要计算 2 加 3,最终 my_add()方法返回计算结果 5。

3.11.2 Python 程序入口

Python 程序的入口方法只有一个,即只有一个文件的__name__等于__main__。如果文件的__name__不等于__main__,则该文件中的代码不会被执行。

一般情况下在文件中需要执行的代码都会写在 if __name__ == "__main__"下面,当文件是被执行文件时,__name__值等于__main__,否则文件__name__值等于文件名。

笔者新建一个 test_name1.py 文件,执行该文件以查看__name__变量的值,代码如下:

```
//第 3 章/new_python/test_name1.py
#加法
def my_add(x, y):
    print("test_name1 文件中__name__的值是:{}".format(__name__))
    return x + y

if __name__ == "__main__":
    add_result = my_add(2, 3)
    print("加法计算的结果是:{}".format(add_result))

#执行结果
test_name1 文件中__name__的值是:__main__
加法计算的结果是:5
```

示例中,在执行结果中会打印出__name__的值为__main__,所以文件中的代码会被执行,加法执行的结果为 5。

笔者再新建一个 test_name2.py 文件,在该文件中导入 test_name1.py 文件中的 my_add()方法进行使用,然后分别打印两个文件中的__name__变量,代码如下:

```
//第 3 章/new_python/test_name2.py
from new_python.test_name1 import my_add

if __name__ == "__main__":
    print("test_name2 文件中__name__的值是:{}".format(__name__))
    add_result = my_add(20, 30)
    print("加法计算的结果是:{}".format(add_result))

#执行结果
test_name2 文件中__name__的值是:__main__
test_name1 文件中__name__的值是:new_python.test_name1
加法计算结果是:50
```

示例中,from…import…表示从…导入…方法,笔者导入了 test_name1.py 文件中的 my_add()方法在 test_name2.py 文件中使用。从执行结果可以看出,test_name2.py 的

__name__变量值等于__main__，test_name1.py的__name__变量值等于文件名，所以test_name1.py的代码没有被执行。

3.11.3 Python模块导入

3.11.2节已经使用了from…import…方式在test_name2.py文件中导入了test_name1.py文件中的my_add()方法。本节讲解如何导入文件和如何导入文件中的方法。

1. 导入模块

导入模块的格式为import 包名.文件名 as 别名。只要读者将模块的路径和模块名编写正确便可以导入模块，导入模块后模块中的方法就可以随便使用了，代码如下：

```
//第3章/new_python/test_import1.py
import new_python.test_name1 as t

if __name__ == "__main__":
    add_result = t.my_add(20, 30)
    print("加法计算的结果是:{}".format(add_result))

#执行结果
加法计算的结果是:50
```

示例中，笔者导入了test_name1模块，并给模块起了个别名t。在执行代码中，笔者使用别名t调用了test_name1中的my_add()方法。当模块中有其他方法时，读者依然可以使用别名t进行调用。

2. 导入模块中的方法

导入模块中的方法的格式为from 包名.模块名 import 方法名1,…,方法名n。当采用这种方式导入时，读者需要提前知道自己要用哪些方法，只要将方法一个一个列出并使用逗号分隔即可，代码如下：

```
//第3章/new_python/test_import2.py
from new_python.test_name1 import my_add, my_sub

if __name__ == "__main__":
    add_result = my_add(20, 30)
    sub_result = my_sub(100, 20)
    print("加法计算的结果是:{}".format(add_result))
    print("减法计算的结果是:{}".format(sub_result))

#执行结果
加法计算的结果是:50
减法计算的结果是:80
```

示例中，笔者先在test_name1.py模块中增加了一个减法方法my_sub()，然后导入了加法方法my_add()和减法方法my_sub()，导入之后直接调用方法名即可。

3.11.4 无参数无返回值方法

无参数无返回值意味着用户不能输入,并且方法执行完成后也不会给用户反馈,这种方法的意义不是很大,笔者在目前的工作中很少使用,代码如下:

```
//第 3 章/new_python/test_method.py
#无参数无返回值的方法
def no_params_no_return():
    print('我是无参数无返回值的方法')

if __name__ == '__main__':
    no_params_no_return()

#执行结果
我是无参数无返回值的方法
```

3.11.5 有位置参数和一个返回的方法

位置参数可以理解为读者调用方法时,传入实际参数的数量和位置都必须和定义方法时保持一致。方法返回值根据需求指定,但必须写在 return 后边,代码如下:

```
//第 3 章/new_python/test_method.py
#有位置参数,有一个返回值的方法
def my_sub(x, y):
    return x - y

if __name__ == '__main__':
    sub_result = my_sub(10, 3)
    print("减法计算的结果是:{}".format(sub_result))

#执行结果
减法计算的结果是:7
```

示例中,笔者新建 my_sub()方法,该方法的参数有两个,即 x、y,方法返回值为 x—y 后的结果。调用 my_sub()方法后笔者传入了两个参数 10 和 3,其中 10 代表 x,3 代表 y。

如果读者使用 my_sub()方法时不传参数或者只传一个参数,Python 就会提示缺少参数,代码如下:

```
//第 3 章/new_python/test_method.py
def my_sub(x, y):
    return x - y

if __name__ == '__main__':
    sub_result = my_sub(10)
    print("减法计算的结果是:{}".format(sub_result))

#执行结果
TypeError: my_sub() missing 1 required positional argument: 'y'
```

示例中，笔者调用 my_sub()方法时只传了一个 10，此时 Python 认为笔者只传了 x 参数没有传 y 参数，所以提示少传了一个参数 y。

3.11.6 有多个返回的方法

在实际工作中，有时用户不仅想知道减法的计算结果，还想知道用户传入的参数是多少，此时就可以使用多个返回值将参数和减法计算的结果都返给用户，代码如下：

```
//第 3 章/new_python/test_method.py
#多个返回值的方法
def my_sub(x, y):
    return x, y, x - y
if __name__ == '__main__':
    x, y, sub_result = my_sub(100, 40)
    print("{} - {} = {}".format(x, y, sub_result))
#执行结果
100 - 40 = 60
```

示例中，笔者在 my_sub()方法中返回了 3 个值，3 个值使用逗号进行分隔。在接收方法返回时，笔者也使用 3 个值进行接收。

3.11.7 默认值参数方法

在实际工作中，有时方法的参数不需要经常变化，此时可以使用默认的参数值对参数先进行赋值，如果参数需要变化，则可在调用时对参数进行重新赋值，代码如下：

```
//第 3 章/new_python/test_method.py
#有默认参数的方法
def my_sub(x = 100, y = 50):
    return x, y, x - y
if __name__ == '__main__':
    x, y, sub_result = my_sub()
    print("{} - {} = {}".format(x, y, sub_result))

#执行结果
100 - 50 = 50
```

示例中，my_sub()方法的参数 x 的默认值为 100，参数 y 的默认值为 50，所以即使在调用 my_sub()方法时不传参数 Python 也不会报错，因为不传参数时 Python 就会按默认值进行操作。

3.11.8 可变参数方法

以加法方法为例，有时用户需要的不是两个数的加法，可能是 n 个数的加法，此时应该怎么处理呢？这时就需要用到可变参数方法，可变参数的意思是用户可以传 1 个参数也可以传 n 个参数，实现方式为在参数变量前加星号，代码如下：

```
//第 3 章/new_python/test_method.py
#可变参数方法
def my_add( * params):
    result = 0
    for item in params:
        result += item
    return params, result

if __name__ == '__main__':
    params, add_result = my_add(1, 2, 3, 4, 5)
    print("{}元组中元素相加的结果为{}".format(params, add_result))

#执行结果
(1, 2, 3, 4, 5)元组中元素相加的结果为 15
```

示例中,my_add()方法的参数 * params 就是可变参数,调用者可以传多个参数进行相加操作。Python 将用户传入的参数 params 当成了一个元组,所以笔者需要遍历元组并对元组中的元素进行相加,最后笔者将 params 和多个参数的相加结果进行了返回。从执行结果可以看出,传入的 5 个参数相加的结果等于 15。

3.11.9 关键字参数方法

关键字参数和可变参数一样,它们都可以让使用者传入 0 个或者 n 个参数。不同之处有两点:第一,关键字参数变量前需要添加两个星号;第二,用户需要传入 key=value 格式的参数,Python 会将用户传入的参数看作一个字典,代码如下:

```
//第 3 章/new_python/test_method.py
#关键字参数方法
def my_add( ** params):
    result = 0
    for value in params.values():
        result += value
    return params, result

if __name__ == '__main__':
    params, add_result = my_add(x = 1, y = 2, z = 3)
    print("{}字典中元素相加的结果为{}".format(params, add_result))

#执行结果
{'x': 1, 'y': 2, 'z': 3}字典中元素相加的结果为 6
```

示例中,笔者对遍历的 params 字典的值进行相加,然后返回 params 和相加结果。从执行结果可以看出,params 是一个字典且 key 和 value 和用户传入的内容一致。

3.11.10 参数的混合使用

学习了多种参数类型,读者只需在编写代码时选择合适的参数类型,不需要使用所有的

参数类型，但如果想要将不同参数混合使用，读者就需要注意参数的顺序。定义方法时参数顺序遵循以下原则：位置参数、默认值参数、可变参数、关键字参数，代码如下：

```
//第 3 章/new_python/test_method.py
#参数的混合使用
def stu_info(name, location = '北京', *years, **work):
    print("姓名:{}".format(name))
    print("居住地:{}".format(location))
    for year in years:
        print("工作年份:{}".format(year))
    for value in work.values():
        print("主要工作内容:{}".format(value))

if __name__ == "__main__":
    stu_info("刘备","蜀", 2022, 2023, work1 = "桃园结义", work2 = "三顾茅庐")

#执行结果
姓名:刘备
居住地:蜀
工作年份:2022
工作年份:2023
主要工作内容:桃园结义
主要工作内容:三顾茅庐
```

示例中，笔者按照位置参数、默认值参数、可变参数、关键字参数的方式进行参数传递，代码可以正常运行。那么如果不按照这个顺序传递参数，代码则会出现什么错误，读者可以自行实验和理解。

3.12 Python 类

除了方法的编程方式外，Python 也支持面向对象的编程方式。面向对象编程是一种程序设计的思想，使用类描述具有相同属性和方法的对象的集合。类是对象的模板，是对象的抽象化，对象是类的实例。

3.12.1 类的定义

Python 中类的定义与方法的定义相似，只不过将 def 关键字换成 class 关键字，不同的是方法的名不需要大写，而类的名需要大写。类中一般包含多个属性和多种方法。

相信大家都有养小动物的经历，那么笔者就使用小动物来举例，定义一个小动物的类，以此来说明类如何编写。笔者从 4 方面入手，第一，小动物类如何定义？第二，小动物类有哪些属性？如小动物的名字、年龄等；第三，小动物类有哪些方法？如小动物会跑、小动物会叫等；第四，小动物类如何使用？即如何实例化小动物类对象，并使用其中的属性和方法，代码如下：

```
//第 3 章/new_python/test_class.py
#类定义
```

```
class Animal():
    #属性
    name = '小黑'
    age = 1
    #方法
    def talk(self):
        return '汪了个汪!'

if __name__ == '__main__':
    animal = Animal()  #实例化
    print("小动物的名字是:{}".format(animal.name))
    print("小动物的年龄是:{}".format(animal.age))
    print("小动物的叫声是:{}".format(animal.talk()))

#执行结果
小动物的名字是:小黑
小动物的年龄是:1
小动物的叫声是:汪了个汪!
```

示例中,笔者解决了刚刚对小动物类如何编写的 4 个疑问,总结如下。

(1) 类的定义:使用 class Animal()定义小动物类。

(2) 类的属性:小动物类有两个属性,分别为名字属性 name 和年龄属性 age。

(3) 类的方法:小动物类有一种方法 talk()。定义类的方法时,方法的第 1 个参数必须是 self,表示实例化后的对象。

(4) 类的实例化:既然说类是对象的模板,那么想要实例化一个小动物就需要按照模板去实例化。代码中的 animal = Animal()就是按照动物类实例化一个小动物对象,并将其赋值给变量 animal。有了 animal 对象后,就可以使用它调用 Animal 类的属性和方法了。

(5) 类实例化后属性和方法的调用:直接使用实例化对象 animal 用“.”方式调用类中的方法和属性,如 animal.name 表示获取小动物的名字。

3.12.2 类的构造方法

上面定义的小动物类中固定了小动物的名字和年龄,这种方式无法适应所有的情况,因为每个小动物不可能都叫小黑,不一定都是 1 岁。如果想解决这个问题,用户就需要在实例化小动物对象时传入小动物的名字和年龄,这就涉及了类的构造方法。

类的构造方法为__init__()方法,该方法默认不传参数,也可以不写,但如果想在类实例化时传入参数,则需要重写构造方法__init__(),代码如下:

```
//第 3 章/new_python/test_class.py
#类定义
class Animal():
    #属性
    name = '小黑'
    age = 1
```

```
    #构造方法
    def __init__(self, name, age):
        self.name = name
        self.age = age
    #方法
    def talk(self):
        return '汪了个汪!'

if __name__ == '__main__':
    xiaobai = Animal('小白', 4)
    print("小动物的名字是:{}".format(xiaobai.name))
    print("小动物的年龄是:{}".format(xiaobai.age))
    print("小动物的叫声是:{}".format(xiaobai.talk()))

#执行结果
小动物的名字是:小白
小动物的年龄是:4
小动物的叫声是:汪了个汪!
```

示例中,笔者编写了小动物类的构造方法__init__()方法,传入了两个参数 name 和 age,这样在类实例化时就需要传入这两个参数,如果不传,则 Python 会报错缺少参数。在类中构造方法的两个参数赋值给了实例化对象的 name 和 age,即 self.name 和 self.age。从执行结果可以看出,实例化对象后对象的 name 和 age 属性是用户传入的值。

3.12.3 类的继承

类的继承可以理解为现实世界中的财产继承,例如 A 继承了父亲的财产,那么 A 就可以使用这些财产。类的继承也是这样的,其主要作用是实现代码的复用。

1. 继承父类

类 B 如果想要继承类 A,则只需在类 B 定义时添加类 A 的名字作为参数,即 class B(A)。这样在使用类 B 时,就可以使用从类 A 中继承来的属性和方法了,其中被继承的类 A 叫作父类,类 B 叫作子类,代码如下:

```
//第 3 章/new_python/test_class.py
#继承
class Animal():
    #属性
    name = '小黑'
    age = 1
    #构造方法
    def __init__(self, name, age):
        self.name = name
        self.age = age
    #方法
    def talk(self):
        return '汪了个汪!'
```

```
class Dog(Animal):
    pass

if __name__ == '__main__':
    dog = Dog('小飞侠', 2)
    print("小狗的名字是:{}".format(dog.name))
    print("小狗的年龄是:{}".format(dog.age))
    print("小狗的叫声是:{}".format(dog.talk()))

#执行结果
小狗的名字是:小飞侠
小狗的年龄是:2
小狗的叫声是:汪了个汪!
```

示例中,Dog 类继承了 Animal 类,即 class Dog(Animal)。笔者在 Dog 类中只写了 pass 的代码,表示什么也不做。子类 Dog 的实例化对象方法同父类一样,即需要输入名字和年龄。实例化后可以使用父类中定义的 name 和 age 属性,也可以使用父类中定义的 talk() 方法,证明子类可以继承并使用父类的属性和方法。

2. 自定义子类

除了可以继承父类的属性和方法外,子类也可以有自己的属性和方法,代码如下:

```
//第 3 章/new_python/test_class.py
#继承
class Animal():
    #属性
    name = '小黑'
    age = 1
    #构造方法
    def __init__(self, name, age):
        self.name = name
        self.age = age
    #方法
    def talk(self):
        return '汪了个汪!'
class Dog(Animal):
    #属性
    legs = 4
    #方法
    def jump(self):
        return '我能大跳!'
if __name__ == '__main__':
    dog = Dog('小飞侠', 2)
    print("小狗的名字是:{}".format(dog.name))
    print("小狗的腿是:{}".format(dog.legs))
    print("小狗的叫声是:{}".format(dog.talk()))
    print("小狗的技能是:{}".format(dog.jump()))
```

```
#执行结果
小狗的名字是:小飞侠
小狗的腿是:4
小狗的叫声是:汪了个汪!
小狗的技能是:我能大跳!
```

示例中,笔者在子类 Dog 中定义了新的属性 legs 和新的方法 jump()。在实际调用过程中,笔者既调用了父类的属性和方法,也调用了子类的属性和方法,执行结果中均可以正常输出。

3.12.4 类的方法重写

如果在继承后发现父类的方法并不能满足子类的需求,则此时可以对方法进行重写。如一个 Cat 类继承了 Animal 类,那么 talk()方法的返回内容就不适用于 Cat 类了,因为猫的叫声应该是“喵”。此时需要对 talk()方法进行重写,代码如下:

```
//第 3 章/new_python/test_class.py
#继承
class Animal():
    #属性
    name = '小黑'
    age = 1
    #构造方法
    def __init__(self, name, age):
        self.name = name
        self.age = age
    #方法
    def talk(self):
        return '汪了个汪!'

class Cat(Animal):
    #方法重写
    def talk(self):
        return '喵了个喵!'

if __name__ == '__main__':
    cat = Cat('小橘', 6)
    print("小猫的叫声是:{}".format(cat.talk()))

#执行结果
小猫的叫声是:喵了个喵!
```

示例中,笔者在子类 Cat 中重写了父类 Animal 的 talk()方法,重新定义了 talk()方法的返回值,所以在 Cat 类实例调用 talk()方法时,执行结果会返回重写方法中的内容。

3.12.5 类的多继承

在实际工作中,读者所编写的类可能需要用到类 A 的方法 1,同时也需要用到类 B 的方法 2,这就需要继承类 A 和类 B,即多继承。继承多个类也很简单,只需用逗号分隔两个父类,即 class S(A, B),代码如下:

```
//第 3 章/new_python/test_class.py
#多继承
class FatherA():
    def func1(self):
        print("父类 A 的方法 1")

class FatherB():
    def func2(self):
        print("父类 B 的方法 2")

class Son(FatherA, FatherB):
    pass

if __name__ == "__main__":
    son = Son()
    son.func1()
    son.func2()

#执行结果
父类 A 的方法 1
父类 B 的方法 2
```

示例中,Son 类继承了 FatherA 类和 FatherB 类,实例化之后可以直接调用 FatherA 类的 func1()方法和 FatherB 类的 func2()方法。

3.13 Python 模块包安装

在实际开发过程中经常会用到 Python 内置模块或第三方模块,例如 time 模块就是内置模块,而 Selenium 模块就是第三方模块。当读者想使用内置模块中的方法时,只需导入模块并进行调用,但如果读者想使用第三方模块中的方法,就需要先安装第三方模块包,安装成功后再进行导入、调用。

3.13.1 pip 安装简介

Python 安装模块包很简单,只要使用 pip 命令即可,但默认安装源是国外网站,由于种种原因有时可能会安装失败或安装时间较长,此时读者需要指定国内的安装源,这样就会加快安装速度。

安装命令格式为 pip install 模块包名,但如果需要指定国内源就需要用到参数 i,并在

参数 i 后面指定国内源地址，见表 3-11。

表 3-11 pip 安装命令简介

命 令	备 注
pip install packagename	安装模块包
pip install packagename -i https://pypi.doubanio.com/simple	豆瓣源安装模块包

3.13.2 PyCharm 命令行安装模块包

在 PyCharm 下方有一个 Terminal 页签，在此页面可以采用 pip 命令的方式进行第三方模块包的安装，如图 3-2 所示。

图 3-2 PyCharm 中 pip 安装

图 3-2 中，笔者在 PyCharm 的 Terminal 页面使用 pip 命令安装 openpyxl 第三方模块，并在命令中使用参数 i 指定了豆瓣源。安装此模块的主要目的是用来操作 Excel。安装完成后，Python 会提示 Successfully installed et-xmlfile-1.1.0 openpyxl-3.0.10，表示安装成功，安装的版本是 3.0.10。

当然，pip 命令除了可以安装第三方模块包以外，还可以卸载第三方模块包、指定特定版本的第三方模块包等，见表 3-12。

表 3-12 pip 安装命令实操

命 令	备 注
pip install openpyxl -i https://pypi.doubanio.com/simple	安装
pip uninstall openpyxl	卸载
pip install openpyxl==3.0.10 -i https://pypi.doubanio.com/simple	指定版本安装

读者可能在安装了很多第三方模块包后想知道自己安装了哪些第三方模块，这就需要用到 pip list 命令。该命令可以看到安装的所有第三方模块包及其版本，命令如下：

```
(MySelenium)E:\workspace\MySelenium>pip list
Package              Version
-------------------  -----------
openpyxl              3.0.10
requests              2.27.1
......
```

在实际工作中安装了很多第三方模块包,并且写了很多代码,如果此时需要将代码给其他同事使用,则同事也需要安装所有的第三方包。此时需要考虑的问题是如何一次性安装这些第三方包呢?方法是将模块包名和版本导出到文件中一次性进行安装。

(1) 将第三模块包导出到文件。

在 PyCharm 中通过 pip freeze > requirements.txt 命令,将工程中用到的所有第三方包导出到 requirements.txt 文件,命令如下:

```
(MySelenium)E:\workspace\MySelenium > pip freeze > requirements.txt
```

(2) 通过文件一次性安装所有的第三方模块包。

pip 命令通过文件安装第三方模块包需要使用 r 参数,表示从文件安装,命令如下:

```
(MySelenium)E:\workspace\MySelenium > pip install -r requirements.txt
```

3.13.3 PyCharm 图形化安装模块包

PyCharm 中除了可以使用命令行安装第三方模块包以外,还可以使用图形化界面进行安装。图形化界面安装需要先找到界面,路径为 File→Settings→Project MySelenium→Project Interpreter。此界面可以查看读者安装了哪些包,功能同 pip list 命令差不多,如图 3-3 所示。

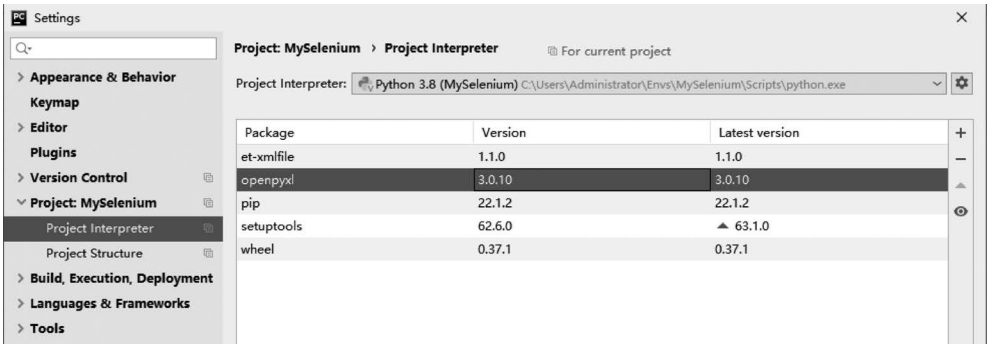

图 3-3 PyCharm 查看安装包

如果读者想安装第三方模块包,则只需单击图 3-3 右侧的加号,进入 Available Packages 窗口。该窗口中读者可以做两件事,设置国内镜像源和安装需要的第三方模块包,如图 3-4 所示。

(1) 设置豆瓣镜像源。

单击 Manage Repositories 按钮,此时会弹出镜像源设置窗口,如图 3-5 所示。在该窗口输入豆瓣源地址 https://pypi.doubanio.com/simple/,保存即可设置成功。

(2) 安装第三方模块包。

设置完豆瓣源后,在 Available Packages 窗口输入 openpyxl,选择需要的模块包,单击

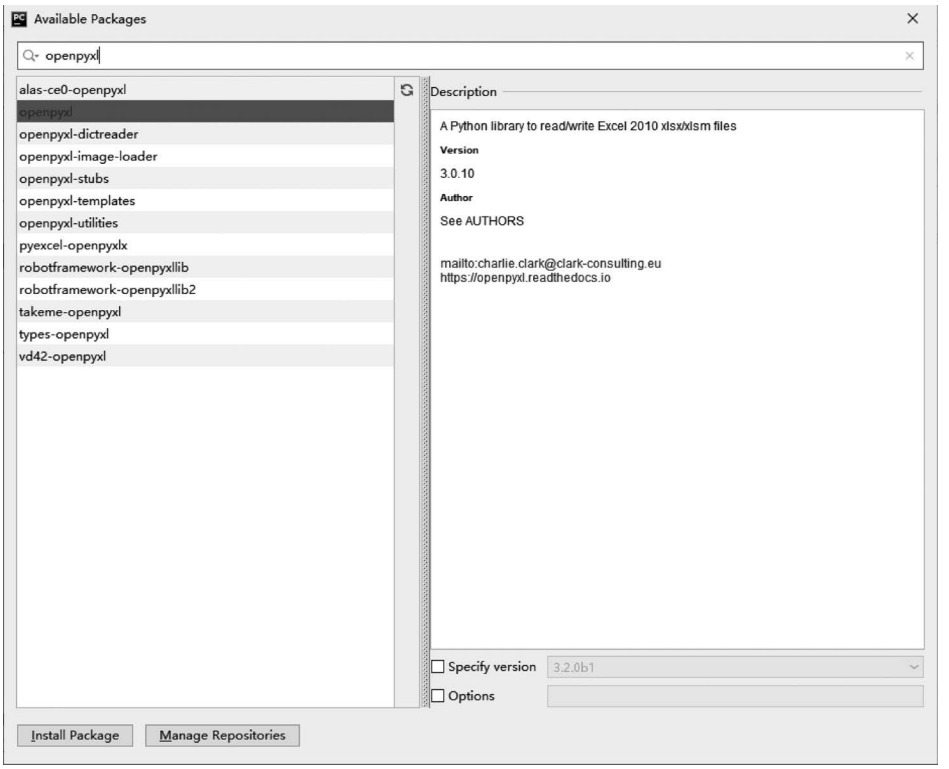

图 3-4 PyCharm 安装第三方模块包

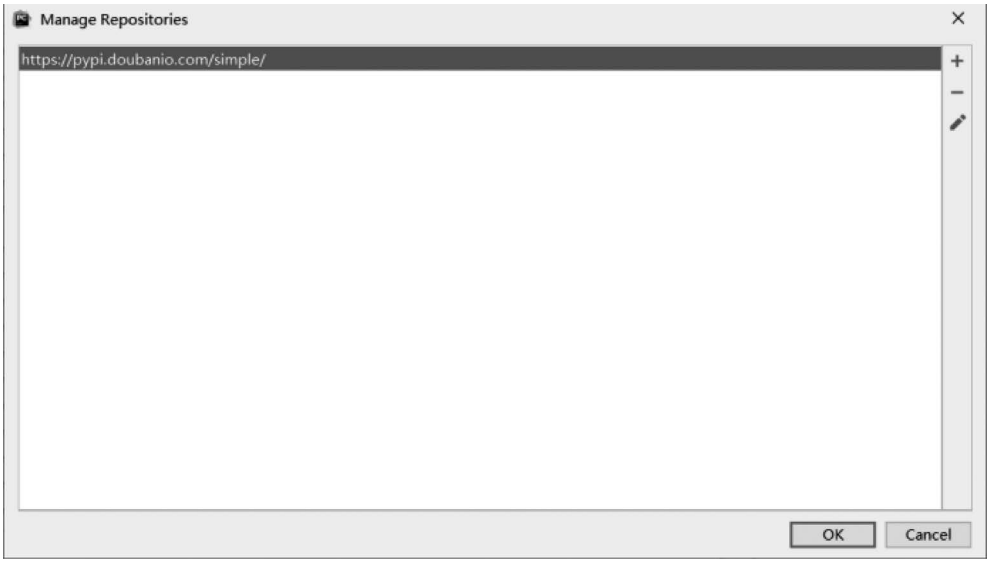

图 3-5 PyCharm 设置豆瓣源

Install Package 进行安装，这样就可以方便地安装好第三方 Excel 操作模块。

在实际工作中笔者还是比较习惯使用 pip 命令进行安装，虽然图形化安装比较方便，但开发人员一般习惯使用 pip 命令进行安装，所以 pip 命令是读者必须掌握的知识。


21min

3.14 Python 的异常

在实际工作中，如果 Python 解析器遇到错误就会停止程序并提示错误信息，这些错误就是 Python 的异常。为了更好地处理异常需要进行异常的捕获，并在发生异常时用更加清晰的语言描述异常，以及使用日志的方式记录异常。

3.14.1 Python 异常捕获

异常捕获的格式为 try-except-else-finally。读者首先需要将可能发生异常的代码写在 try 语句下方，然后在 except 语句下方编写发生异常时需要执行的代码，在 else 语句下方编写的代码在不发生异常时会被执行，finally 语句的意思是无论是否发生异常都需要执行其下方代码。示例代码如下：

```
try:
    print("可能发生异常的代码")
except Exception as e:
    #发生异常时捕获
    print(e)
else:
    print("没有异常时执行的代码")
finally:
    print("无论是否有异常都会执行的代码")
```

3.14.2 Excel 操作及异常捕获

笔者在安装第三方模块时安装了 openpyxl 模块，该模块是用来操作 Excel 文件的，接下来笔者将使用该模块来演示异常捕获。

笔者新建两个文件 parse_excel.py 和 UICases.xlsx，其中 parse_excel.py 文件的内容是笔者封装的 Excel 解析类，UICases.xlsx 文件是一个包含用例的 Excel 文件。

1. ParseExcel 类

笔者首先导入 openpyxl 模块的 load_workbook()方法，使用该方法传入 filename 参数即可获得 excel 实例对象。ParseExcel 类暂时仅简单地实现实例化 excel 对象功能，代码如下：

```
//第 3 章/new_python/test_excel1.py
from openpyxl.reader.excel import load_workbook

class ParseExcel():
```

```
    book = ''
    def __init__(self, excelPath):
        self.book = load_workbook(filename = excelPath)
```

示例中，self.book 就是实例化后的 excel 对象。

2. 传入不存在的文件

在上述代码中，正确传入已存在的 Excel 文件不会报错，但如果尝试传入不存在的 Excel 文件，则系统会报错并提示文件不存在，代码如下：

```
//第 3 章/new_python/test_excel1.py
from openpyxl.reader.excel import load_workbook

class ParseExcel():
    book = ''
    def __init__(self, excelPath):
        self.book = load_workbook(filename = excelPath)

if __name__ == '__main__':
    my_excel = ParseExcel('./UICases2.xlsx')

#执行结果
FileNotFoundError: [Errno 2] No such file or directory: './UICases2.xlsx'
```

示例中，笔者传入了一个不存在的 Excel 文件 UICases2.xlsx。从执行结果可以看出，代码会产生 FileNotFoundError 异常，即文件不存在。

3. 异常捕获

接下来笔者将使用 try-except 来捕获上一小节的异常，捕获到异常以后用中文打印“Excel 文件不存在!”，这样会让用户更加清楚发生了什么，代码如下：

```
//第 3 章/new_python/test_excel2.py
from openpyxl.reader.excel import load_workbook

class ParseExcel():
    book = ''
    def __init__(self, excelPath):
        try:
            self.book = load_workbook(filename = excelPath)
        except FileNotFoundError as e:
            print('Excel 文件不存在!')
            print(e)

if __name__ == '__main__':
    my_excel = ParseExcel('./UICases2.xlsx')

#执行结果
Excel 文件不存在!
[Errno 2] No such file or directory: './UICases2.xlsx'
```

示例中,已知实例化 Excel 的代码可能会发生异常,所以读者将其放在 try 语句下方,在 except 语句下方笔者不仅打印了自定义提示,还打印了系统的异常提示 e,从执行结果可以看出,两个提示信息均被打印,从用户的角度来看自定义的中文提示更容易理解。

4. 传入错误类型文件

如果读者细心一点就会发现除了文件不存在异常可能存在外,文件类型不正确的异常也可能存在。例如传一个 Word 文件就应该会出现文件类型不正确异常,代码如下:

```
//第 3 章/new_python/test_excel1.py
from openpyxl.reader.excel import load_workbook

class ParseExcel():
    book = ''
    def __init__(self, excelPath):
        self.book = load_workbook(filename = excelPath)

if __name__ == '__main__':
    my_excel = ParseExcel('./UICases2.docx')

#执行结果
openpyxl.utils.exceptions.InvalidFileException: openpyxl does not support .docx file format,
please check you can open it with Excel first. Supported formats are: .xlsx,.xlsm,.xltx,.xltm
```

示例中,读者将传入文件的扩展名改成 docx,表示传入一个 Word 文件。此时 Python 会提示文件类无效,即 InvalidFileException。

5. 捕获多个异常

既然现在知道可能会出现两个异常,那么就需要捕获这两个异常,可以使用两个 except 语句来分别捕获不同的异常,代码如下:

```
//第 3 章/new_python/test_excel3.py
from openpyxl.reader.excel import load_workbook
from openpyxl.utils.exceptions import InvalidFileException

class ParseExcel():
    book = ''
    def __init__(self, excelPath):
        try:
            self.book = load_workbook(filename = excelPath)
        except FileNotFoundError as e:
            print('Excel 文件不存在!')
            print(e)
        except InvalidFileException as e:
            print('文件类型错误!')
            print(e)

if __name__ == '__main__':
    my_excel = ParseExcel('./UICases2.doc')
```

```
#执行结果
文件类型错误!
openpyxl does not support .doc file format, please check you can open it with Excel first.
Supported formats are: .xlsx,.xlsm,.xltx,.xltm
```

示例中,笔者又使用 except 捕获了 InvalidFileException 异常,捕获到异常后打印“文件类型错误!”提示。需要注意的是,捕获该异常时需要导入异常类,否则 Python 会报错。

6. 捕获所有异常

在实际工作中不可能对一个个异常进行捕获,一方面是书写麻烦,另一方面也不可能考虑到所有异常,所以需要一种能够捕获所有异常的方法,代码如下:

```
//第 3 章/new_python/test_excel4.py
from openpyxl.reader.excel import load_workbook

class ParseExcel():
    book = ''
    def __init__(self, excelPath):
        try:
            self.book = load_workbook(filename = excelPath)
        except Exception as e:
            print('Excel 文件加载时出现错误!')
            print(e)

if __name__ == '__main__':
    my_excel = ParseExcel('./UICases2.xlsx')

#执行结果
Excel 文件加载时出现错误!
[Errno 2] No such file or directory: './UICases2.xlsx'
```

示例中,笔者直接捕获 Exception,这样无论发生哪种类型的异常 Python 都会进行捕获,从而达到不会遗漏异常的目的。

7. 解析 excel 类

最后笔者使用 openpyxl 模块对常用的操作 Excel 文件的代码进行简单封装,读者可以根据自己的需要进行适当修改和优化,代码如下:

```
//第 3 章/new_python/test_excel5.py
from openpyxl.reader.excel import load_workbook

class ParseExcel():
    #属性
    excelPath = ''
    book = ''
    sheet = ''
    #初始化
```

```
    def __init__(self, excelPath):
        self.excelPath = excelPath
        self.book = load_workbook(filename = excelPath)
    #根据 Sheet 页名字获取 Sheet 页
    def getSheetByName(self, sheetName):
        self.sheet = self.book.get_sheet_by_name(sheetName)
        return self.sheet
    #Sheet 页最大行数
    def getMaxRow(self):
        return self.sheet.max_row
    #Sheet 页最大列数
    def getMaxColumn(self):
        return self.sheet.max_column
    #获取 Sheet 页某个单元格的值
    def getCellValue(self, rowNum, columnNum):
        return self.sheet.cell(row = rowNum, column = columnNum).value
    #设置 Sheet 页某个单元格的值
    def setCellValue(self, rowNum, columnNum, value):
        try:
            self.sheet.cell(row = rowNum, column = columnNum).value = value
        except Exception:
            print('写入单元格内容时出错!')
        else:
            self.saveExcel()
    #保存
    def saveExcel(self):
        self.book.save(self.excelPath)
    #关闭
    def closeExcel(self):
        self.book.close()

if __name__ == '__main__':
    my_excel = ParseExcel('./UICases.xlsx')
    my_excel.getSheetByName('流程')
    #遍历流程页中的内容
    rows = my_excel.getMaxRow()
    columns = my_excel.getMaxColumn()
    print("Excel 的流程页中共 {} 行、{} 列".format(rows, columns))
    for i in range(1, rows):
        for j in range(1, columns):
            print("第 {} 行,第 {} 列".format(i, j))
            print("内容是:{}".format(my_excel.getCellValue(i, j)))
    my_excel.closeExcel()
```

示例中,笔者对获取 Sheet 页、Sheet 页最大行数、Sheet 页最大列数、获取 Sheet 页的值、设置 Sheet 页的值、保存 Excel 和关闭 Excel 代码进行了封装,读者在后期的工作和学习中可以对其进行异常捕获优化,并直接应用到实际工作中。

3.15 装饰器

装饰器可以理解为在不破坏原方法的基础上对现有方法的功能进行拓展，装饰器的使用是在被装饰方法上方添加“@装饰器名”。本节只对装饰器进行简单介绍，目的是当在开发过程中遇到装饰器时，读者可以有个基本的认识，不至于在开发过程中产生疑惑。

3.15.1 不使用装饰器

例如笔者已经写了 100 种方法，老板要求算出每种方法的耗时情况，此时该如何解决这一问题？在不使用装饰器的情况下，笔者只能在每种方法的第 1 行记录开始时间，在每种方法的最后一行记录结束时间，然后打印出结束时间减开始时间所得到的耗时，代码如下：

```
//第 3 章/new_python/test_decorator1.py
import time

def add(x, y):
    start_time = time.time()
    time.sleep(3)
    add_result = x + y
    end_time = time.time()
    print("加法方法耗时:{}秒".format(end_time - start_time))
    return add_result

if __name__ == "__main__":
    add_result = add(2, 2)
    print("加法计算结果:{}".format(add_result))

#执行结果
加法方法耗时:3.0005931854248047s
加法计算结果:4
```

示例中，笔者在加法 add()方法的第 1 行获取了开始时间，在方法代码结束后获取了结束时间，然后打印了结束时间减开始时间，即 add()方法运行耗时，其中 time.sleep(3)表示让程序休眠 3s，目的是避免耗时统计出现结果为 0 的情况，所以执行结果中 add()方法的执行时间是 3s 以上。

上述方法确实可以计算出每种方法的耗时，但如果 100 种方法都需要计算耗时，则修改每种方法会带来很大的工作量，并且如果添加代码出错，则会影响原有方法的正常运行。

3.15.2 无参装饰器

3.15.1 节的办法虽然能解决问题，但需要耗费很长时间且影响原有代码的质量，所以不是解决问题的最好方式。接下来笔者将自定义装饰器解决上述问题。

1. 无参装饰器格式

无参装饰器指的是在使用该装饰器方法时不能传递参数。无参装饰器是最简单的装饰器,格式如下:

```
#无参数装饰器
def 装饰器名(func):
    def wrapper(*args, **kwargs):
        #额外功能代码
        res = func(*args, **kwargs)
        #额外功能代码
        return res
    return wrapper
```

示例中,参数 func 表示被装饰的方法,装饰器返回的是一个 wrapper 方法,用专业术语来讲叫作闭包,简单来讲就是在方法内嵌套方法。读者在封装装饰器时,只需套用以上格式,并在被装饰方法的前后添加需要的代码。

2. 无参装饰器封装

根据需求笔者需要封装一个装饰器,用来计算所有方法的耗时,思路就是在方法的前边记录开始时间,在方法的后边记录结束时间,并打印结束时间减开始时间,代码如下:

```
//第3章/new_python/test_decorator2.py
import time

def take_time(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        time.sleep(3)
        res = func(*args, **kwargs)
        end_time = time.time()
        print("方法的执行时间是:{}秒".format(end_time-start_time))
        return res
    return wrapper
```

示例中,笔者根据无参装饰器的格式封装了一个 take_time()方法,以此计算耗时,读者可以关注以下几点。

(1) 定义装饰器名 take_time。

(2) 在被装饰方法 func()之前记录开始时间 start_time,并休眠 3s,即 time.sleep(3)。

(3) 在被装饰方法 func()之后记录结束时间 end_time,最后打印耗时 end_time-start_time。

3. 无参装饰器的使用

笔者封装好计算耗时的装饰器后,将该装饰器应用到加法方法上,看装饰器是否可以计算出执行加法运算的耗时,代码如下:

```
//第 3 章/new_python/test_decorator3.py
from new_python.test_decorator2 import take_time

@take_time
def add(x, y):
    add_result = x + y
    return add_result

if __name__ == "__main__":
    add_result = add(2, 2)
    print("加法的计算结果:{}".format(add_result))

#执行结果
方法的执行时间是:3.0000338554382324s
加法的计算结果:4
```

示例中,笔者首先导入了计算耗时装饰器 take_time,并在加法方法上应用了计算耗时装饰器,格式为@take_time。从执行结果中可以看出,装饰器方法计算除了加法方法的耗时,同时也正确地计算出了加法的结果。

接下来如果想解决问题,则只需在所有方法的上方都使用 take_time 装饰器,这样既不会影响原有方法,又可以快速满足需求。

3.15.3 有参装饰器

在 3.15.2 节中,每种方法都休眠 3s,如果笔者想让用户传入参数来自定义休眠时间,则该如何实现? 实现方法就是使用有参装饰器,让用户传入休眠时间。

1. 有参装饰器格式

有参装饰器无非就是在无参装饰器外面又包了一层方法定义,为内部的装饰器提供所需的参数,格式如下:

```
#有参装饰器
def 装饰器名(param):
    def decorator(func):
        def wrapper(*args, **kwargs):
            #额外功能代码
            res = func(*args, **kwargs)
            #额外功能代码
            return res
        return wrapper
    return decorator
```

2. 有参装饰器封装

根据有参装饰器的格式,笔者封装了一个有参装饰器以让用户输入方法休眠时间,代码如下:

```
//第 3 章/new_python/test_decorator4.py
import time

def take_time(delay = 1):
    def decorator(func):
        def wrapper( * args, ** kwargs):
            start_time = time.time()
            time.sleep(delay)
            res = func( * args, ** kwargs)
            end_time = time.time()
            print("方法的执行时间是:{}秒".format(end_time - start_time))
            return res
        return wrapper
    return decorator
```

示例中,笔者定义了有参装饰器 take_time()方法,并且参数休眠时间的默认值为 1,表示如果使用者不传参数,则休眠 1s。

3. 有参装饰器的使用

封装好自定义休眠时间装饰器后,笔者再次将其应用在加法方法上,代码如下:

```
//第 3 章/new_python/test_decorator5.py
from new_python.test_decorator4 import take_time

@take_time(2)
def add(x, y):
    add_result = x + y
    return add_result

if __name__ == "__main__":
    add_result = add(2, 2)
    print("加法的计算结果:{}".format(add_result))

#执行结果
方法的执行时间是:2.0008039474487305s
加法的计算结果:4
```

示例中,笔者使用 take_time 装饰器并传入参数 2,表示想休眠 2s。从执行结果可以看出,加法执行结果确实是 2s 多。

3.16 Python 多线程

多线程可以简单地理解为程序可以同时执行多个不同的任务。在学习多线程之前,读者应该简单了解一下进程和线程的概念。通俗来讲,运行一个程序就是一个进程(主线程),线程是进程中负责执行程序的最小单位,一个进程中至少包含一个线程。正常情况下程序是按照顺序一条条命令执行的,如果想在打游戏的同时听歌,就需要使用多线程。

多线程的执行时是由 CPU 进行调度的，CPU 采用时间切片技术调度多个线程，每个时间片分配给一个线程执行，当时间片用完将停止该线程的执行，同时将进入下一个时间片运行下一个线程。由于时间片很短，所以在使用者看来多个线程是同时进行的。

3.16.1 创建线程

Python 提供了一个 threading 模块，读者可以通过 threading 模块中的 Thread()方法新建线程，每个线程可以通过 target 参数指定执行不同的方法，代码如下：

```
//第 3 章/new_python/test_threading1.py
import threading
import time

def task():
    print("线程开始!")
    for i in range(3):
        time.sleep(2)
        print(i)
    print("线程结束!")

if __name__ == "__main__":
    th1 = threading.Thread(target = task)
    th1.start()
    print("进程结束!")

#执行结果
线程开始!
进程结束!
0
1
2
线程结束!
```

示例中，笔者使用 threading.Thread()方法来新建线程 th1，target 参数指明了该线程执行 task()方法，然后通过 start()方法执行线程。从执行结果中可以看出，线程 th1 可以正常执行，但存在一个问题，即在线程结束运行之前进程已经结束，这不符合笔者的预期，笔者的目标是线程结束运行后进程再结束。

3.16.2 join()方法

在 3.16.1 节中进程提前结束的问题可以通过 threading 中的 join()方法解决。该方法所做的工作就是等待子线程执行完，即让主线程进入阻塞状态，一直等待其他的子线程执行结束后，主线程再终止。简单来讲就是等待子线程结束后再执行主线程后续的命令，代码如下：

```
//第 3 章/new_python/test_threading2.py
import threading
```

```
import time

def task():
    print("线程开始!")
    for i in range(3):
        time.sleep(2)
        print(i)
    print("线程结束!")

if __name__ == "__main__":
    th1 = threading.Thread(target = task)
    th1.start()
    th1.join()
    print("进程结束!")

#执行结果
线程开始!
0
1
2
线程结束!
进程结束!
```

示例中,笔者增加了 th1.join()方法,从执行结果可以看出,Python 先执行子线程直到结束,然后执行主线程中剩下的语句,满足了笔者的需求。

3.16.3 线程方法传参

当线程执行的方法需要传参时,可以在 threading 模块的 Thread()方法中添加参数 args,该参数是一个元组,当传多个参数时需要使用逗号分隔,但当只传一个参数时需要在参数后边加上逗号,代码如下:

```
//第 3 章/new_python/test_threading3.py
import time
import threading

def add(a, b):
    print("加法开始!")
    result = a + b
    print("加法的计算结果为{}".format(result))
    time.sleep(1)
    print("加法结束!")

if __name__ == "__main__":
    th1 = threading.Thread(target = add, args = (4, 4))
    th1.start()
    th1.join()
    print("进程结束!")
```

```
#执行结果
加法开始!
加法的计算结果为 8
加法结束!
进程结束!
```

示例中,线程 th1 执行加法方法,使用 args 传入两个数字进行相加,执行结果中两个数相加结果正确。

3.16.4 创建多个线程

在 3.16.1 节中只介绍了如何新建一个线程,但实际上可能需要新建几个线程。笔者将使用循环的方式来新建线程,并将新建的线程放到列表中,然后遍历列表并调用线程的 start()方法和 join()方法,代码如下:

```
//第 3 章/new_python/test_threading4.py
import time
import threading

def add(a, b):
    print("加法开始!")
    result = a + b
    print("加法的计算结果为{}".format(result))
    time.sleep(1)
    print("加法结束!")
def sub(a, b):
    print("减法开始!")
    result = a - b
    print("减法的计算结果为{}".format(result))
    time.sleep(1)
    print("减法结束!")

if __name__ == "__main__":
    fun_dict = {"add":(1,1), "sub":(10,2)}
    threads_list = []
    #新建子线程
    for key,value in fun_dict.items():
        key = eval(key)
        th = threading.Thread(target = key, args = value)
        threads_list.append(th)
    #启动子线程
    for thread in threads_list:
        thread.start()
    for thread in threads_list:
        thread.join()
    print("进程结束!")
```

```
#执行结果
加法开始!
加法的计算结果为 2
减法开始!
减法的计算结果为 8
加法结束!
减法结束!
进程结束!
```

示例中,笔者将线程需要执行的方法定义为字典 fun_dict,字典元素的 key 为执行方法的名字;value 为执行方法的参数且格式为元组。

有了线程需要执行的方法名和参数字典后,笔者使用 3 个循环进行操作。第 1 个循环遍历字典 fun_dict 新建线程,其中 eval()方法是 Python 的内置方法,其作用是去掉字符串前后的双引号,遍历的过程中将新建的线程添加到线程列表 threads_list 中;第 2 个循环遍历列表 threads_list,并调用 start()方法启动每个线程;第 3 个循环还是遍历列表 threads_list,并调用 join()方法阻塞主线程。

此处笔者需要解释下为什么进行两次遍历,而不是 1 次遍历。如果 1 次遍历既启动线程又阻塞主线程,则代码会一直等到第 1 个线程代码执行结束后再执行第 2 个线程的代码,这并不是笔者想要的效果,所以先遍历一次以启动所有线程,然后遍历一次以阻塞主线程。读者可以自行尝试上述两种做法,从执行结果中很容易看出执行的顺序。

3.17 本章总结

虽然本章讲解了大量的 Python 知识,但相比专门讲 Python 开发的书籍,本章内容还有许多地方需要细化,还有很多内容没有讲解。学习本章 Python 知识的目的是能应对接下来的 UI 自动化测试开发,如果读者想要开发自动化测试平台,则还需要继续深入学习 Python Web 相关知识。